Optimization and parallelization of the boundary element method for the wave equation in time domain

Berenger Bramas

université de BORDEAUX

THÈSE

PRÉNTÉ À

L'UNIVERSITÉ DE BORDEAUX I

ÉCOLE DOCTORALE DE MATHÉMATIQUES ET D'INFORMATIQUE

Par **BRAMAS Bérenger**

POUR OBTENIR LE GRADE DE

DOCTEUR

SPÉCIALITÉ : INFORMATIQUE

Optimisation et Parallèlisation de la Méthode des Élements Frontières pour l'Équation des Ondes dans le Domaine Temporel

---

Soutenue le : 15 Février 2016

Après avis des rapporteurs :

BIROS George	Professeur - The University of Texas at Austin
VUDUC Richard	Professeur (Associé) - Georgia Institute of Technology

Devant la commission d'examen composée de :

BIROS George	Professeur - The University of Texas at Austin	Rapporteur
COULAUD Olivier	Directeur de Recherche - Inria Bordeaux - Sud-Ouest	Directeur de Thèse
HAV'E Pascal	Chercheur - IFP Energies nouvelles	Examinateur
LANTERI Stephane	Directeur de Recherche - Inria Sophia Antipolis	Examinateur
NAMYST Raymond	Professeur - Université de Bordeaux	Président du jury
SYLVAND Guillaume	Chercheur - Airbus Group Innovations	
TERRASSE Isabelle	Directrice de Recherche - Airbus Group	Examinateur
VUDUC Richard	Professeur (Associé) - Georgia Institute of Technology	Rapporteur

– 2016 –

Optimization and Parallelization of the Boundary Element Method for the Wave Equation in Time Domain

Bérenger Bramas

in fulfillment of the requirements for the degree of
Doctor in the subject of
Computer Science

February 15th, 2016

Referees	George Biros	Professor - The University of Texas at Austin	
	Richard Vuduc	Associate Professor - Georgia Institute of Technology	
Committee	George Biros	Professor - The University of Texas at Austin	Referee
	Olivier Coulaud	Research Director - Inria Bordeaux - Sud-Ouest	Advisor
	Pascal Havé	Researcher - IFP Energies nouvelles	Examiner
	Stephane Lanteri	Research Director - Inria Sophia Antipolis	Examiner
	Raymond Namyst	Professor - The University of Bordeaux	Committee president
	Guillaume Sylvand	Researcher - Airbus Group Innovations	
	Isabelle Terrasse	Research Director - Airbus Group	Examiner
	Richard Vuduc	Associate Professor - Georgia Institute of Technology	Referee

université de BORDEAUX

Bérenger Bramas– ©2016
Doctoral School of Mathematics and Computer Science,
Department of Computer Science,
Inria Bordeaux - Sud-Ouest, Université de Bordeaux 1.
Generated the Thursday 18th February, 2016 at 10:47

Optimisation et Parallélisation de la Méthode des Éléments Frontières pour l'Équation des Ondes dans le Domaine Temporel

Bérenger Bramas

Résumé

La méthode des éléments frontières pour l'équation des ondes (BEM) est utilisée en acoustique et en électromagnétisme pour simuler la propagation d'une onde avec une discrétisation en temps (TD). Elle permet d'obtenir un résultat pour plusieurs fréquences à partir d'une seule résolution. Dans cette thèse, nous nous intéressons à l'implémentation efficace d'un simulateur TD-BEM sous différents angles. Nous décrivons le contexte de notre étude et la formulation utilisée qui s'exprime sous la forme d'un système linéaire composé de plusieurs matrices d'interactions/convolutions. Ce système est naturellement calculé en utilisant l'opérateur matrice/vecteur creux (SpMV). Nous avons travaillé sur la limite du SpMV en étudiant la permutation des matrices et le comportement de notre implémentation aidé par la vectorisation sur CPU et avec une approche par bloc sur GPU. Nous montrons que cet opérateur n'est pas approprié pour notre problème et nous proposons de changer l'ordre de calcul afin d'obtenir une matrice avec une structure particulière. Cette nouvelle structure est appelée une matrice tranche et se calcule à l'aide d'un opérateur spécifique. Nous décrivons des implémentations optimisées sur architectures modernes du calcul haute-performance. Le simulateur résultant est parallélisé avec une approche hybride (mémoires partagées/distribuées) sur des nœuds hétérogènes, et se base sur une nouvelle heuristique pour équilibrer le travail entre les processeurs. Cette approche matricielle a une complexité quadratique si bien que nous avons étudié son accélération par la méthode des multipoles rapides (FMM). Nous avons tout d'abord travaillé sur la parallélisation de l'algorithme de la FMM en utilisant différents paradigmes et nous montrons comment les moteurs d'exécution sont adaptés pour relâcher le potentiel de la FMM. Enfin, nous présentons des résultats préliminaires d'un simulateur TD-BEM accéléré par FMM .

Mots-clés : calcul haute performance, programmation parallèle, optimisation, vectorisation, GPU, méthode des éléments frontières, équation des ondes, acoustique, électromagnétisme.

Optimization and Parallelization of the Boundary Element Method for the Wave Equation in Time Domain

Bérenger Bramas

Abstract

The time-domain BEM for the wave equation in acoustics and electromagnetism is used to simulate the propagation of a wave with a discretization in time. It allows to obtain several frequency-domain results with one solve. In this thesis, we investigate the implementation of an efficient TD-BEM solver using different approaches. We describe the context of our study and the TD-BEM formulation expressed as a sparse linear system composed of multiple interaction/convolution matrices. This system is naturally computed using the sparse matrix-vector product (SpMV). We work on the limits of the SpMV kernel by looking at the matrix reordering and the behavior of our SpMV kernels using vectorization (SIMD) on CPUs and an advanced blocking-layout on Nvidia GPUs. We show that this operator is not appropriate for our problem, and we then propose to reorder the original computation to get a special matrix structure. This new structure is called a slice matrix and is computed with a custom matrix/vector product operator. We present an optimized implementation of this operator on CPUs and Nvidia GPUs for which we describe advanced blocking schemes. The resulting solver is parallelized with a hybrid strategy above heterogeneous nodes and relies on a new heuristic to balance the work among the processing units. Due to the quadratic complexity of this matrix approach, we study the use of the fast multipole method (FMM) for our time-domain BEM solver. We investigate the parallelization of the general FMM algorithm using several paradigms in both shared and distributed memory, and we explain how modern runtime systems are well-suited to express the FMM computation. Finally, we investigate the implementation and the parametrization of an FMM kernel specific to our TD-BEM, and we provide preliminary results.

Keywords: high-performance computing, parallel programming, optimization, vectorization, GPU, boundary element method, wave equation, acoustic, electromagnetism.

Optimisation et Parallélisation de la Méthode des Éléments Frontières pour l'Équation des Ondes dans le Domaine Temporel

Bérenger Bramas

Résumé Étendu

La méthode des éléments frontières dans le domaine temporel (BEM) en acoustique et en électromagnétisme est utilisée pour simuler la propagation des ondes avec une discrétisation en temps (TD). Elle permet d'obtenir le résultat de plusieurs fréquences en une seule résolution. Dans cette thèse, nous étudions l'implémentation d'une simulateur TD-BEM efficace sous différents angles. Nous décrivons le contexte de notre étude et la formulation utilisée qui peut s'exprimer par un système linéaire creux composé de plusieurs matrices d'interactions/convolutions. L'objectif est de calculer l'état des inconnus a^n au temps n, en utilisant les matrices de convolution M^k et le vecteur d'illumination l^n qui décrit l'impact de l'onde incidente sur le maillage au temps n, par

$$a^n = (M^0)^{-1} \left(l^n - \sum_{k=1}^{K_{max}} M^k \cdot a^{n-k} \right) . \tag{1}$$

Ce système est naturellement calculé en utilisant le produit matrice vecteur creux (SpMV) sur les K^{Max} matrices M^k.

Dans cette étude, nous nous intéressons dans un premier temps aux limites du SpMV, de la permutation des matrices jusqu'à l'implémentation d'un noyau efficace sur CPU ou GPU. Dans un simulateur industriel, le surcoût d'un algorithme de permutation doit être moindre à comparé du gain pour le noyau SpMV. Nous montrons qu'une heuristique gloutonne pour graphe a un coût faible et améliore le stockage des matrices creuses par bloc. Néanmoins, les blocs sont composés à plus de 50% de zéro, et les performances du SpMV réduisent donc du même facteur. Cette observation est la motivation principale pour regarder notre problème d'un point de vue différent.

Dans la formulation originale, la boucle externe est sur les K^{Max} matrices, et les boucles inférieures sur les lignes et les colonnes de chaque matrice creuse. Nous proposons de réordonner ce calcul pour obtenir des matrices à la structure particulière car composées d'un vecteur non-nul par ligne; la boucle externe se fait alors sur les colonnes des matrices creuses. Afin d'avoir un accès mémoire approprié, les matrices creuses sont remplacées par une nouvelle structure appelée matrices en tranche. Nous étudions l'opérateur correspondant pour utiliser ces nouvelles matrices, et nous montrons que calculer plusieurs pas de temps simultanément augmente le ratio du nombre d'opération par mot mémoire par rapport au SpMV. Toutefois, cet opérateur est très sensible aux optimisations. Nous décrivons notre implémentation sur CPU utilisant la vectorisation en C mais aussi en assembleur avec pour objectif de réduire le nombre d'accès et d'augmenter l'utilisation des

registres mémoires. Nous décrivons ensuite un algorithme de parallélisation hybride pour notre simulateur et démontrons que la factorisation/résolution de M^0 devient critique si on augmente le nombre de nœud.

Nous étendons notre simulateur pour fonctionner sur les architectures hybrides composées de plusieurs CPU/GPU. Le calcul des matrices en tranche sur GPU est exigeant, et il est obligatoire d'utiliser une approche par bloc avec des optimisations à la compilation. Nous comparons deux structures de bloc et leurs noyaux respectifs sur GPU, et nous introduisons une nouvelle heuristique pour équilibrer le travail entre les unités de calculs hétérogènes pendant les premières itérations. Nous validons notre simulateur sur 5 noeuds chacun composés de 24 CPU et 4 GPU, et nous démontrons que nous avons une bonne accélération lorsque le problème passe en mémoire dans les GPU.

Néanmoins, cette approche matricielle a une complexité quadratique pour la résolution mais aussi pour la construction des matrices. Cette complexité peut être limitante quand à la taille du problème que nous pouvons résoudre et nous incite à étudier un simulateur TD-BEM avec une accélération par la méthode des multipoles rapides.

Nous nous intéressons dans un premier temps a l'algorithme de la FMM et à sa parallélisation avec plusieurs paradigmes en mémoire partagée et distribuée. La parallélisation de la FMM est simple avec une approche *fork-join* comme celle proposée par OpenMP. D'un autre côté, nous montrons comment les moteurs d'exécution sont bien adaptés pour vraiment exprimer les dépendance de la FMM. De plus, nous montrons comment améliorer les peformances grâce à la propriété commutative des opérations de la FMM. Nous proposons une nouvelle structure de donnée appelée arbre-en-groupe pour paralleliser la FMM avec un moteur d'exécution. Cet arbre-en-groupe permet de parametrer la granularité des tâches et potentiellement de réduire le nombre de dépendances. Nous introduisons une nouvelle FMM à base de tâches en mémoire distribuée grâce à notre arbre-en-groupe. Nous comparons les comportements de différentes implémentations en mémoire partagée et distribuée pour des simulations de particules. Les différents développements logiciels sont inclus dans la bibliotheqèe ScalFMM qui est un package libre pour la FMM.

Enfin, nous étudions l'implémentation d'un solver TD-BEM utilisant la FMM et un noyau FMM approprié. Nous calculons les interactions entre les feuilles en utilisant l'approche matricielle et les matrices d'interactions, et nous utilisons la FMM pour calculer les interactions lointaines à l'aide de fonctions définies sur la sphère unité. Cela nous permet de ne pas générer toutes les matrices d'interactions. Toutefois, cette approche par FMM peut être implémentée de différentes façons et doit être correctement parametrée. Les différents opérateurs de la FMM peuvent être calculés dans le domaine temporel ou fréquentiel, ce qui requiert des transformés de Fourier supplémentaires. Nous montrons qu'il est plus intéressant de rester dans le domaine temporel si l'on utilise une méthode d'interpolation appropriée pour effectuer le décalage temporel. De plus, nous n'avons pas besoin de calculer la FMM en entière à chaque itération si l'on prend en compte le temps mis par les ondes pour se propager.

Contents

I dedicate this thesis to my wife Dorothée & my soon Orlando for their support and their patience. I also dedicate this dissertation to my family and especially to my parents who pushed me in this path.

Introduction

The time-domain boundary-element method (TD-BEM) for the wave equation is used to simulate the propagation of an acoustic or electromagnetic wave on a surface mesh. More precisely, it simulates how an incident wave emitted by a source propagates and reflects over the discretization elements of the mesh with a progression in time. A conversion of the time-domain results with a Fourier transform allows us to study several frequencies, which makes a TD-BEM solver appropriate to study wide-band applications. Nevertheless, the frequency-domain BEM (FD-BEM) has been more studied than the time-domain formulations [1]. The size of the discretization elements decreases proportionally with the frequency studied and thus the number of unknowns to cover the mesh increases quadratically. The use of supercomputers and high-performance computing (HPC) is necessary to compute large simulations from a memory and workload standpoints.

The TD-BEM formulation used in this study has been introduced in [2] where a problem is expressed as a linear system with a summation stage naturally computed using sparse matrix-vector products (SpMV). However, this approach has two main limitations. First, it is well known that the SpMV is a memory-bound operation which achieves a small percentage of the peak performance on most processing units. Secondly, the generation of the matrices and the solve of the system have a quadratic complexity relatively to the number of discretization elements of the mesh which may limit the size of the problems we are able to compute. In this thesis, we investigate how to address these two issues and to provide an efficient TD-BEM solver for modern HPC architectures. The resulting application aims at replacing an existing TD-BEM software based on the SpMV.

Scope of study. The hardware structure of the processing units makes the development of efficient SpMV extremely challenging because of the low ratio of number of operations against size of the data. In our case, by reordering the computation, we obtain special matrices with one dense vector per row which increases this ratio. Therefore, we have focused on the development of an efficient kernel for such matrices on CPU and GPU. The resulting operator is not a general SpMV, but is specific to our problem. However, the development and the management of the hardware particularities are classic and could be applied in various other development. The description of a fast multipole method (FMM) kernel to accelerate our TD-BEM has been proposed in [3]. We have studied different parallelization schemes for the general FMM algorithm and use them above our implementation of TD-BEM FMM kernel. This kernel has been implemented with high level optimizations but it is still a specific formulation for our TD-BEM.

Manuscript organization. The first chapter provides some details of our TD-BEM formulation for acoustic and describes the nowadays HPC architectures and their relative challenges. The related works are developed in the second chapter; it approaches the SpMV operators with its

optimized implementations and the general FMM algorithm. The last three chapters describe our contributions and present numerical and performance studies. In chapter 3, we concentrate on the matrix-approach of the TD-BEM. Then, in chapter 4, we study the FMM parallelization strategies in shared and distributed memory. In the last chapter, we focus on our TD-BEM accelerated by FMM and give preliminarily results. Finally, in annexes we provide the material related to the underlying mathematics and algorithms of our different implementations.

Acknowledgement: this work has been supported by the Airbus Group, Inria and Conseil Régional d'Aquitaine initiative.

1

Problem Statement

In this chapter, we give an overview of the context and the background for the different fields that are connected to our work. We first recall that our development is done inside an industrial context and we describe the aim of the wave propagation simulations. Then, we give the core formulation of our Time-Domain Boundary Element Method (TD-BEM) expressed with the sparse matrix-vector product (SpMV). We finish by summarizing the nowadays HPC problematics to explain the current challenges in the development of HPC applications.

1.1 Industrial Application Context.

This work has been done in collaboration with Airbus Group Innovations, an entity of Airbus Group devoted to research and development for the usage of Airbus Group divisions (Airbus Civil Aircraft, Airbus Defence & Space, Airbus Helicopters). For more than 20 years, the numerical analysis team has been working on integral equations and boundary element methods for wave propagation simulations. The resulting software solutions are daily basis in acoustics for installation effects computation, aeroacoustic simulations (in a coupled scheme with other tools), and in electromagnetism for antenna siting, electromagnetic compatibility or stealth. Since 2000, these frequency-domain Boundary Element Method (BEM) tools have been extended with a multipole algorithm (called fast multipole method) that allows us to solve very large problems, with tens of millions of unknowns, in reasonable time on parallel machines. More recently, $\mathcal{H}$-matrix techniques have enabled the design of fast direct solvers, able to solve problems with millions of unknowns for a very high accuracy without the usual drawback associated with the iterative solvers: no control on the number of iterations, difficulty to find a good pre-conditioner, etc. At the same time, Airbus Group Innovations works on the design and on the optimization of the time-domain BEM (TD-BEM). The resulting applications presented in this study are a layer of an industrial computational work-flow and must be robust enough to be used industrially. We delegate to some existing black-boxes the management of the mesh or the generation of the data to concentrate on

their computation. Different choices, in terms of algorithms or external libraries, are restricted by the industrial end-usage.

BEM for the Wave Equation Applications. The studies of wave propagations in acoustics and electromagnetism have numerous applications. In aeronautical engineering, they are used to design the aircraft but also to guarantee their robustness. For example, the position of the motors is important because it changes the volume of the sound heard inside the cockpit. Besides, the cockpit of an airplane is mainly composed of metal and all the communications rely on electromagnetic signals such that it is crucial to ensure that the messages send to the engine/wing controls or the communication with the ground are correct. The interferences between the components must be avoided and the reaction to external sources like lightning must be accounted for.

As we clarify in the next sections, the objects are discretized in order to study the problem on small faces/elements. In Figure 1.1, we show a *mesh* example of an airplane with three antennas.

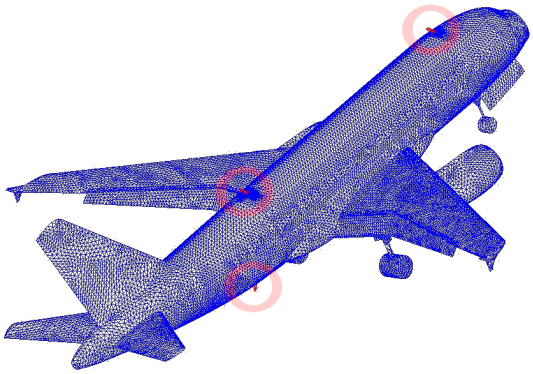
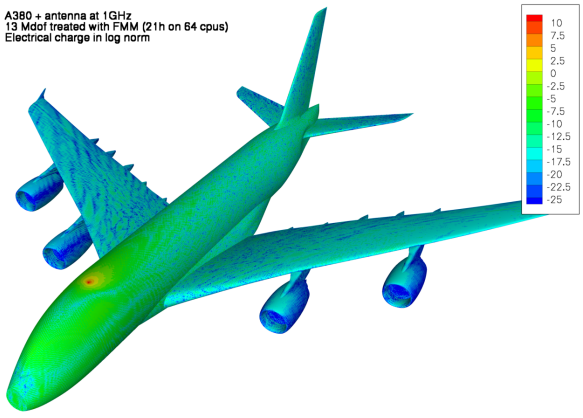
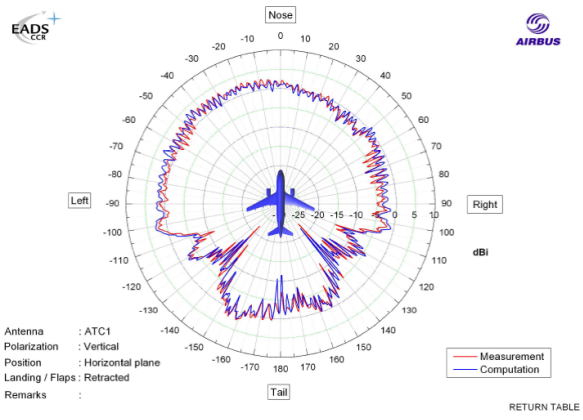


Figure 1.1: Discretization of an airplane in a mesh.

In Figure 1.2a and Figure 1.2b, we show two results from electromagnetic simulations. The first is used to study the impact of an antenna on the cabin and the second shows a comparison between a simulations and results obtained with real measures.



(a) Result from a simulation to study the emission of an antenna



(b) Comparison of the wave emission obtain numerically by simulation or with real measures

Figure 1.2: Electromagnetism study examples.

1.2 Time-Domain Boundary Element Method (TD-BEM) for the Wave Equation

Many physical processes can be described by partial differential equations (PDE) but only few special cases can be solved analytically. In order to overcome this shortcoming, many numerical methods have been developed like the BEMs, the FEMs or the spectral methods. The Boundary Element Method (BEM) is a numerical computational method which solves these PDEs when they are formulated as integral equations (i.e. in a boundary integral equation form - BIE). These methods are developed for boundary integral equations that result when boundary value problems on spatial domains are transformed to integral equations on the boundary of the physical domain. In the BEM, the $3D$ geometry of the problem is discretized in space by $2D$ surfacic mesh elements to describe only the boundary of the original object [4]. The mathematical details of our problem formulation are presented in Appendix A, but it is not a prerequisite to the understanding of our contribution and the implementation parts. Readers who feel concerned by the physical aspects and the underlying mathematical formulation may read the complete description that leads to the linear system expression. The given problem description and formulation were taken from [3] and the time-domain formulation was originally introduced in [2] for electromagnetism. The resulting linear system for the Maxwell equations in electromagnetism is similar to the one presented here for the acoustic. Therefore, the computational methods that are proposed in this study can be easily adapted to electromagnetism.

1.2.1 Linear System Formulation

In this section, we review the key-points of the formulation of our TD-BEM and focus to its linear system expression. An incident acoustic plane wave w with a velocity c and a wavelength λ is emitted on a boundary Ω. The surface Ω is discretized by a classical finite-element method using triangular elements with the unknowns/degrees of freedom located at the vertices. We denote by N the total number of unknowns in the system. The wave equation is also discretized in time with a step Δt, and the number of time step T has to be chosen from the frequency range of the study and the size of the object considered. The illumination vector l^n contains the influence of the incident wave w over the unknowns of the mesh at iteration time $t_n = n\Delta t$. Once the wave illuminates the location where the unknowns are defined, it is then reflected by them over the mesh. This complex behavior is represented numerically by a set of interaction/convolution matrices M^k, $0 \leq k \leq K^{max}$.

The aim is to compute the state of the unknowns a^n at time n using the convolution matrices M^k and the vector l^n, which describes the incident wave emitted on the unknowns of the mesh at time n, by

$$\sum_{k\geq 0}^{K^{max}} M^k \cdot a^{n-k} = l^n \,. \tag{1.1}$$

The vectors a and l are of dimension N and the matrices M of dimension $N \times N$ with N the number of unknowns. The terms $\sum_{k=1}^{K^{max}} M^k \cdot a^{n-k}$ represent the past taken into account using the previous states of the unknowns defined by the vectors a^{n-k} with $k > 0$. This equation expresses the fact that what happens now at time n (a^n) is the consequence of what happened in the past $n-k$ (a^{n-k})

on the elements far from $k \cdot c \cdot \Delta t$. This distance relation is accounted for in the matrices M^k.

The original Equation (1.1) can be rewritten as in formula (1.2) where the left-hand side is the state to compute, and the right-hand side is known from the K^{max} previous time steps and the test case definition. For T iterations/time step $n > 0$, we compute

$$a^n = (M^0)^{-1} \left(l^n - \sum_{k=1}^{K^{max}} M^k \cdot a^{n-k} \right) . \tag{1.2}$$

1.2.2 Interaction/Convolution Matrices

The matrix M^k contains the interactions between unknowns that are separated by a distance around $k \cdot c \cdot \Delta t$ and contains zero for unknowns that are closer or further than this distance. They have the following properties:

- The number of non-zero values for a given matrix M^k depends on the structure of the mesh (the distance between the unknowns) and the physical properties of the system c, λ and Δt. In realistic configurations, the matrices M^k are sparse because the elements far from $k \cdot c \cdot \Delta t$ are only a small part of the mesh.

- The matrices M^k are zero for $k > K^{max} = 2 + \ell_{max}/(c\Delta t)$, with ℓ_{max} the diameter of the object ($\ell_{max} = max_{(x,y)\in\Gamma\times\Gamma}(|x-y|)$).The waves propagate at velocity c and what happens on Γ at time t will not impact the results after the date $(t + L/c)$.

- M^0 is almost diagonal since it represents the interaction of the elements with themselves and their close neighbors.

- The position of the non-zero values in the matrices is driven by the numbering of the unknowns and with an appropriate numbering, we expect the matrices to look like Figure 1.3.

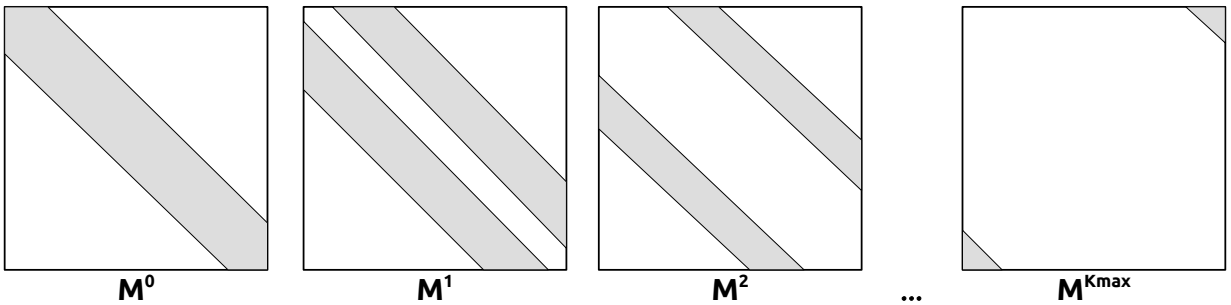
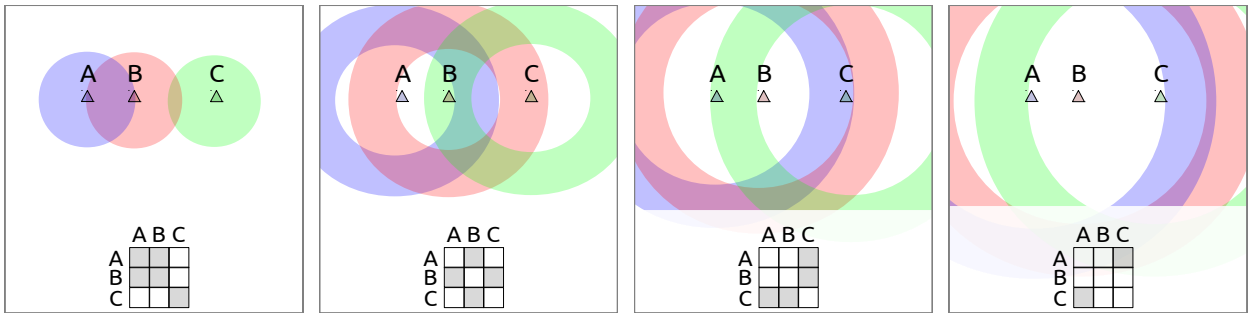


Figure 1.3: Expected shape of M^k matrices

Figure 1.4 illustrates the construction of these matrices and shows where the non-zero (NNZ) values are depending on the delay taken by a wave emitted by an unknown to pass over another one. Two NNZ values are contiguous in a row, for example $M^k(i,j)$ and $M^k(i,j+1)$, if the unknowns j and $j+1$ are both at a distance $k.c.\Delta t$ from i. On the other hand, two NNZ values are contiguous in a column, for example $M^k(i,j)$ and $M^k(i+1,j)$, if the unknowns i and $i+1$ are both at a distance $k.c.\Delta t$

from j. Therefore, numbering consecutively the unknowns that are spatially close is a way among others to increase the chance to have contiguous values in the interaction matrices. However, with a few exceptions, it is not possible to find a numbering of the unknowns which leads to perfect NNZ diagonals for a $3D$ problem.



(a) M^0 (b) M^1 (c) M^2 (d) M^3

Figure 1.4: Example of M^k matrices for three unknowns A, B, C in $2D$. A wave emitted from each unknown is represented at every time step. When a wave is around an unknown, a value is added in the matrix which is symbolized by a gray square. All matrices M^k with $k > 3$ are zero since the highest distance between elements is $\leq 3c\Delta t$.

In our simulations, the meshes are fixed (constant in time) and this gives important properties to the matrices and the way of addressing the problem; the matrices are the same at each time step, they are in finite number and the same matrix M^0 is solved at each time step.

1.2.3 Resolution Algorithm

The solution is computed in two steps. In the first step, the past is taken into account using the previous values of a^p with $p < n$ and the interaction matrices as shown in Equation (1.3). The result s^n is subtracted from the illumination vector, see Equation (1.4).

$$s^n = \sum_{k=1}^{K^{max}} M^k \cdot a^{n-k}, \tag{1.3}$$

$$\widetilde{s}^n = l^n - s^n. \tag{1.4}$$

In the second step, the state of the system at time step n is obtained after solving the following linear system

$$M^0 a^n = \widetilde{s}^n. \tag{1.5}$$

The first step is the most expensive part from a computational standpoint. The solution of Equation (1.5) is extremely fast, since matrix M^0 is symmetric, positive definite, sparse and almost diagonal and we can solve it using a sparse direct solver for example. Figure 1.5 represents graphically an iteration of the solve.

We refer to the process of computing s^n as the summation stage and it has to be done at each time step n from 1 to T. It seems natural to compute this operation using K^{max} SpMVs between the interaction matrices M^k and the past values of the unknowns a^{n-k} with $1 \leq k \leq Min(n, K^{max})$.

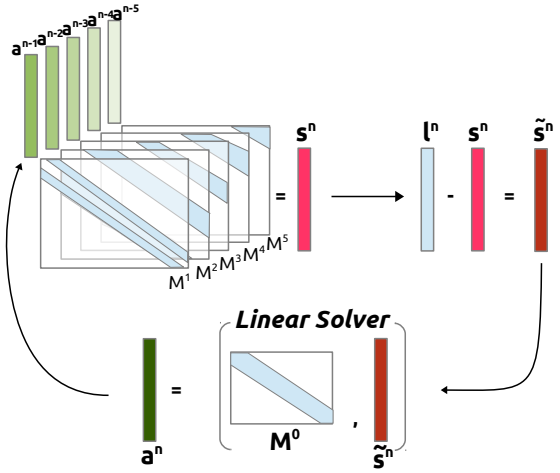
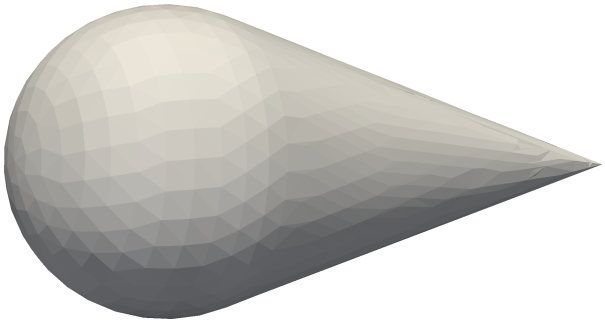


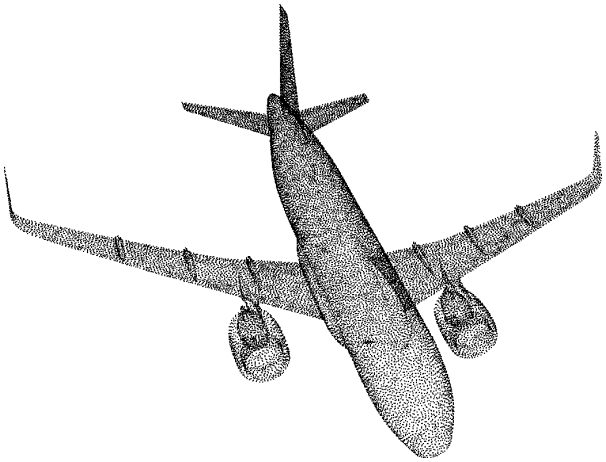
Figure 1.5: Solve algorithm schematic view

1.2.4 Study Test Cases

In our study, the mesh is discretized using triangular elements with the unknowns/degrees of freedom located at the vertices in most cases. When required, Gaussian quadrature numerical technique is used to evaluate integral over these elements. In the present manuscript, we validate our methods using two test cases: the cone-sphere cases are constructed by aggregating half a sphere with a cone, as shown in Figure 1.6a, while the airplane is an industrial test case presented in Figure 1.6b.



(a) Cone-sphere mesh example.



(b) Airplane mesh example.

More precisely, we use several variants of the cone-sphere with different number of unknowns but with the same size of triangles/discretization elements and wave properties. Therefore, the size of the mesh (in space) increases with the number of unknowns. The specifications of the test cases are given in Table 1.1 where we provide some information related to the linear systems.

From these information, we can estimate the cost of the simulations in terms of floating-point operations and their memory footprint. If we consider that the solution step of the system associated with M^0 has the cost of a matrix-vector product, the total amount of floating-point operations for the entire airplane simulation is $130\,651 \times 10^9$. In addition, it takes $50\,GB$ to store the complete

Case	C-927	C-4269	C-10012	C-22468	Airplane
Number of unknowns	927	4269	10012	22468	23962
Number of interaction matrices M^k (K^{max})	117	244	370	551	341
Number of NNZ in the interaction matrices	7.5×10^6	1.5×10^8	8.6×10^8	4.3×10^9	5.5×10^9
Number of time steps (T)	2033	4345	6647	9957	10823
Number of RHS	[1,2]	[1,2]	[1,2]	[1,2]	1

Table 1.1: Specifications of the cone-sphere and airplane cases.

airplane data when working in single precision. Therefore, the use of HPC is mandatory to solve large problems in a reasonable time.

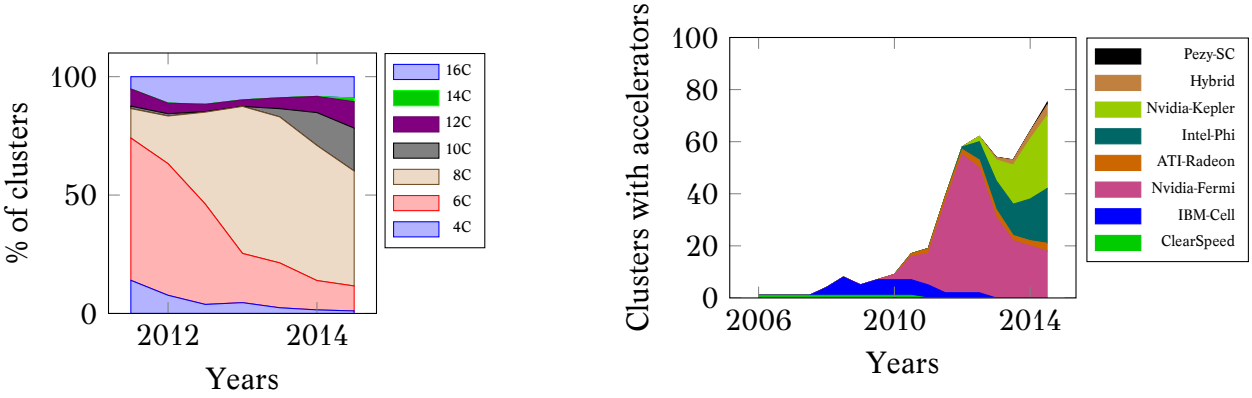
1.3 Modern HPC

The High-Performance Computing (*HPC*) is a large domain at the cross between numerous fields. We state that HPC is related to mainly two problematics, namely solving problems as large as possible and/or solving problems as fast as possible. So forth the HPC is used for the development of scientific applications and non-scientific applications. The first category deals with scientific problems coming from different origins, and it is common to use the name of Scientific Computing to include them all. It is connected to modeling, physics, applied mathematics and numerical analysis among others. On the other hand, we find in the second category all the software that do not imply numerical problems, and we could refer to network applications, databases, visualization tools and all the server modules for example. Its related sub-fields are: parallel computing, high-performance programming, software engineering and algorithms.

1.3.1 Parallel and High Performance Computing

Nowadays, the computer ecosystem is composed by a high diversity of hardware, which makes the development for HPC more challenging but also more exciting. In Figure 1.7, we show statistics taken from the famous TOP-500 [5] which illustrate the constant changes in the fastest supercomputers. It shows that the number of cores inside a node is variable and that there is no real dominant type even if the evolution shows that nodes are fit out with more cores per processor over the time in Figure 1.7a. For less than ten years, accelerators are becoming a growing part of HPC, and we can see in Figure 1.7b that not only there is a high diversity of accelerators, but also some of them have already disappeared after only a few years of presence in the top. Moreover, the desktop computers should not be forgotten because it is usual for many users to solve small problems on their personal machines. These machines are less diverse compared to super-computers; they are mainly composed by 4 cores and 1 graphic card which might be used as an accelerator.

Thus, developing optimized kernels for a given architecture is one of the major problematics because it requires to adapt the algorithms and to develop data structures that are well-suited. For example, the memory hierarchy on CPU or the usage of SIMD can cause significant changes in the way of addressing a problem. Then, the parallelization i.e. the division of the work, its distribution and the load balancing above the available resources is necessary. As a final objective,



(a) Number of cores per processor

(b) Clusters with accelerators

Figure 1.7: TOP500 [5] statistics (may not be perfectly accurate because extracted manually)

HPC applications should be designed to work on a different kind of clusters and to survive to the technology evolution by being easy to upgrade and to maintain. Since most server applications are request-oriented, they focus more on concurrency than parallelism by trying to proceed independent tasks/applications or requests and to balance them among the servers and their CPUs. Whereas scientific applications are more parallel with an objective of finishing the earliest with a dedicated number of resources. In this domain, the more common languages are *Fortran*, *C* and *C++* which are all compiled languages.

Parallelization. The shared memory parallelization, over the CPU cores of a node, is usually done using a native OS thread library like POSIX or an HPC oriented library like the compiler implementations of the OpenMP [6] standard. Under the hood, the OpenMP implementations can rely on the POSIX threads, but they propose a different interface and some facilities in the work division. Based on these libraries, it is common to parallelize HPC scientific applications with the fork-join paradigm using parallel loop or tasks and barrier synchronizations; the work is parallelized stage by stage with an explicit barrier between each of them. MPI implementations [7] are the dominant tools to parallelize in distributed memory, between nodes connected by a network, and most of the applications developed in the last decades are made above hybrid MPI/OpenMP.

Addressing the development of efficient kernels on accelerators is not an easy task and while the OpenCL [8] standard is supported by the majority of accelerators, using specific language and optimizations is still required. For example, an OpenCL kernel can be executed on a CPU or a GPU but using a shared memory buffer might be a drawback on CPU since this role is already supported by the hardware cache. From our experience, CUDA [9] is the most appropriate language to work on NVidia GPUs whereas OpenCL seems well-suited to develop on AMD/ATI GPUs or Intel Xeon Phi. The plug-in of these kernels inside a parallel application is easy to do by hand with the dedication of one thread/core per accelerator but the balancing, and the memory transfer can be extremely tricky and technical. Moreover, it is difficult to be both optimized, by having a good scalability, and generic, by being efficient on different cluster configurations. Runtime systems try to address this problem because they give an abstraction of the machine and dissociate the

algorithm, the kernels and the hardware with the help of advanced schedulers for some of them. The different available runtime systems might not support the same operations, and in addition they are not targeting the same architectures. State-of-the-art linear algebra libraries have been developed at the top of runtime systems and show the potential of such tools [10; 11; 12; 13].

Work balancing. Balancing the workload among different processing units which have distinct performance capacities is an ongoing research. The problem is even more difficult for irregular applications because it is challenging to have an accurate performance model and to be able to split the workload appropriately. The two major solution classes are the division of the work, by assigning one part of the work to each process depending on its performance capacities, and the distribution of the work, by generating multiple tasks and scheduling them among the processing units.

In the division approaches, pure static balancing can be appropriate as in [14] where authors show interesting results on Sparse Grid Interpolation but this it is not applicable in most of the cases. When the problem is iterative, it is possible to improve the division at each iteration; starting from a certainly unbalanced initial guess, the resulting execution time on each processing unit drives the next division. As an example, such a system is used in [15] where they divide the work between the CPUs and the GPUs for the Linpack benchmark, and they use the result of an iteration to better balance the next one. One can see this kind of algorithm as a mix between static and dynamic balancing.

Dynamic balancing is related to the distribution approaches, and it is mostly naturally implemented using task programming; the whole problem is split in pieces of pertinent granularity, and then the processing units consume the tasks. But then the original issue is transferred to the scheduling strategy which should dispatch the work correctly and hide the data movement.

1.3.2 Numerical Operations in Scientific Computing

The fact that numerical operations constitute the costly part of scientific applications must be taken into account to achieve performance. The problems to solve have different origins but the most common are from physics: molecular dynamics, fluid dynamics, wave equations, weather prediction, quantum mechanics, etc. Thus scientific computing is at the cross of applied mathematics (numerical methods, numerical analysis, etc.) and modeling [16]. Any improvement in one of the layers may signify important changes in the others: the complexity decrease of an algorithm, a new implementation of a kernel or a new hardware are some examples that might lead to rewrite completely an application.

Floating-point operations. From the theoretical definition of a numerical algorithm, we can estimate the number of floating-point number instructions based on the different problem parameters. Such instructions/operations are called *Flop* for floating-point operations, and we consider that on modern computer, the time to perform additions/subtractions or multiplications/divisions is about the same. Thus the *Flop* cost of a problem can be found for any resolution without even implement-

ing an application by counting each elementary operation that are needed to solve it. However, the implementation choice or the architecture where the program is executed may change the way of counting. For example, some mathematical functions, like cosine or exponential, can be done by software using several *Flop* or by special hardware modules in fewer instructions. Moreover, modern architectures propose a Floating Multiple-Add (FMADD) instruction to realize two elementary *Flop* $a+ = b * c$ and in the literature, it is common to count one FMADD as a single *Flop*. Among the implementation choices, we can refer to the extra merge that could be required when we divide the work between threads and which may change the real number of *Flop* that have been performed during an execution. In addition to theoretical estimations, one can rely on hardware counters to know precisely the number of *Flop* in an execution which is a nice asset to study low level optimizations. Finally totaling up these operations provides insight into which parts of the algorithm are most time-consuming and how computation time increases as the system gets larger.

However, in this study we only count the *Flop* from the theoretical algorithm or specify explicitly when doing differently. We do not count the extra *Flop* coming from implementation choices or optimizations and when using mathematical operators, we count them using their cost on standard CPUs. In our description the number of *Flop* is hardware and implementation independent. From the *Flop* cost of a problem and the execution wall time, we get a speed in *Flop* per second ($Flop/s$ or $GFlop/s$) that allow us to compare architectures even if some have better instruction sets or costly optimizations. It gives us an rough idea of the number of the Floating-Point instructions that have been performed which might be different from the hardware *Flop* counters.

2

Background and State of the Art

In this chapter, we present different sparse-matrix storages and describe the state-of-the-art implementation of the SpMV. In a second part, we describe the fast multipole method (FMM) algorithm. Finally, we give a brief overview of some others time-domain BEM formulation and we finish by introducing the outline of our contributions.

2.1 Sparse Matrix Vector Product (SpMV)

The linear system from the formulation presented in Section 1.2 can be implemented using sparse matrix-vector product (SpMV). A famous definition of what makes a matrix sparse has been given in [17]:
"The matrix may be sparse, either with the non-zero elements concentrated on a narrow band centered on the diagonal or alternatively they may be distributed in a less systematic manner. We shall refer to a matrix as dense if the percentage of zero elements or its distribution is such as to make it uneconomic to take advantage of their presence".
In other words, if there is any advantage by exploiting the zeros, for example, by saving time or memory, then the matrix should be considered as sparse. Therefore, from this definition the sparse aspect is related to the context and the numerical algorithm which is the SpMV in our case. The optimization of the SpMV operator has been widely studied because this is an essential operation in many scientific applications and the costly part of sparse iterative solvers. However, removing the zeros of a matrix leads to new storages and new computational kernels and while the gain of using a sparse matrix instead of a dense one can be huge in terms of memory occupancy and speed; the effective *Flop* rate of a sparse kernel is generally remaining low compared to its dense counterpart. In fact, in a sparse matrix storage, we provide a way to know the respective column and row of each non-zero value (NNZ). Therefore, the general SpMV is a bandwidth/memory bound operation because it pays the price of this extra storage and makes it having a low ratio of *Flop* to perform against data occupancy. Moreover, the difficulties are amplified by the hierarchical

memory on modern CPUs and the multi-streaming design of the GPUs.

Researches in optimizing the SpMV have been done for decades and had give two main orientations. The first includes the matrix storage and the way the values are accessed and computed. The second part is about reordering the matrices and providing an appropriate matrix access pattern which can benefit to the storage. Finally, to achieve performance and to remove some hardware bound, it is required to focus on the specificity of the treated problem: providing an efficient SpMV for all kinds of matrices is clearly difficult, but implementing a SpMV for matrices with a specific and structured pattern is much more feasible.

In this section, we introduce some well-known sparse matrix storages and describe their memory requirement and some of their advantages/disadvantages on CPU. We give an overview of the hardware limits and difficulty, and we describe some reordering techniques.

CPU Hardware Consideration

The CPU architecture and its evolution have driven the past research on the SpMV. In the appendix C we give a reminder of the specificities but recall here the key-points that should be in mind when looking at SpMV storages. Current CPUs have hard memory constraints, mostly from the hierarchical aspect with different latencies and bandwidths depending on the location of the requested data. Therefore, the efficiency of an execution is very sensitive to the temporal and spatial localities. Finally, the speed - in terms of instruction per second - of the CPU is quite high and have been improved much more than the memory which becomes the bound when the ratio of *Flop* per word is low. This has been well studied in various papers as in [18; 19] and it has been shown that there is a need of register blocking, cache blocking, and if possible multiplication by multiple vectors.

2.1.1 Sparse Matrix Storages Survey

Despite the numerous different storages, we focus on the most important and include some storages for specific patterns. A sparse matrix storage represents how the NNZ values of a matrix are stored in memory but also how they are acceded during the SpMV computation. Therefore, when we refer to a storage and its quality we might also refer to the resulting performance of the SpMV.

Coordinate (COO) or Element-by-element (EBE) Storage

The more natural way to store a sparse matrix is to use a triple for each NNZ value composed by the real NNZ value and its original position in the matrix. Such storage is called COO (coordinate) or EBE (element-by-element) and is illustrated in Figure 2.1. Therefore, for each value we store two integer indexes and one floating point value, which gives the memory occupancy $S_{COO} = N_{NNZ} \times (2S_i + S_f)$ with N_{NNZ} the number of NNZ, S_i the size of an index (usually an integer of 4 bytes) and S_f the size of a floating point value (4 or 8 bytes). Using this storage, the memory occupancy can be large and the memory access pattern can be irregular because it is governed by the matrix rows/columns ordering and the order of access of the values. However, for a very low amount of

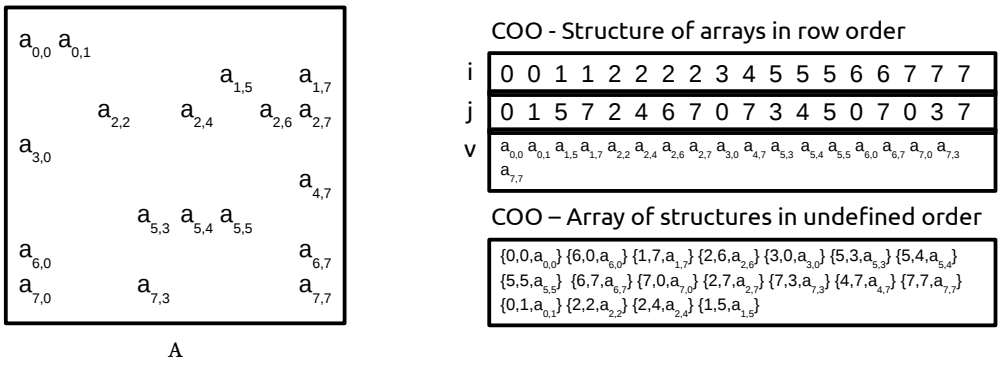


Figure 2.1: COO/EBE matrix storage with one triple per NNZ using multiple arrays or array of structures.

NNZ if, for example, most of the rows/columns do not even have a single NNZ, this storage is appropriate and may give above-average performance. Finally, it is useful to save a matrix or to convert it to another format since it is easy to sort or permute.

Compressed Row Storage (CRS) or Compress Sparse Row (CSR) Storage

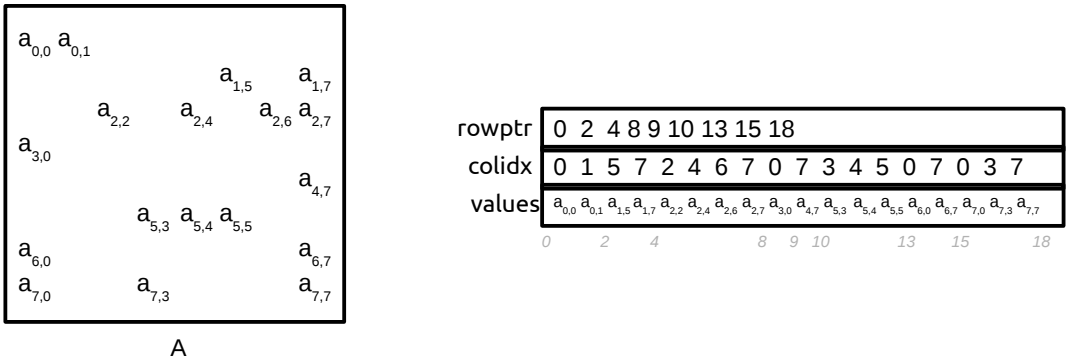


Figure 2.2: CRS matrix storage: the *rowptr* array gives the offset position of the NNZ in the *colidx* and *values* arrays, these two arrays contains the values and their corresponding column indexes in row major.

The CRS is a well-known storage and is used as a de-facto standard in SpMV studies. The main idea of this storage is to avoid storing individual row indexes for each NNZ value but instead counts the number of values that each row contains. Figure 2.2 presents an example of the CRS storage. The NNZ values of the original matrix are stored into a *values* array in row major (one row after the other) and in column ascending order. In a secondary array *colidx* we store the column indexes of the NNZ values in the same order. Finally, *rowptr* contains the positions of the NNZ in *values* for each row: the row i has NNZ from index $rowptr[i]$ to $rowptr[i+1]$. This storage uses less memory than the COO if the number of NNZ is greater than the number or rows which is the more common situation. During the computation, the values are read one row after the other making the access to the result vector linear and potentially unique. Moreover, the input vector is read from left to right at each row computation. The data occupancy is given by $S_{CRS} = N_{NNZ} \times (S_i + S_f) + S_i \times (nb_{row} + 1)$ with nb_{row} the number of rows in the original matrix. Reordering/permuting the matrix may improve the spatial locality by having only small jumps in

the column-direction between values of the same row. This storage also exists with a compression in the columns called CCS for Compressed Column Storage. Its memory occupancy and data locality have been studied in details in various papers as in [20].This storage does not target a particular matrix pattern which makes it appropriate in general.

Fixed-size Block Storage (FSB)

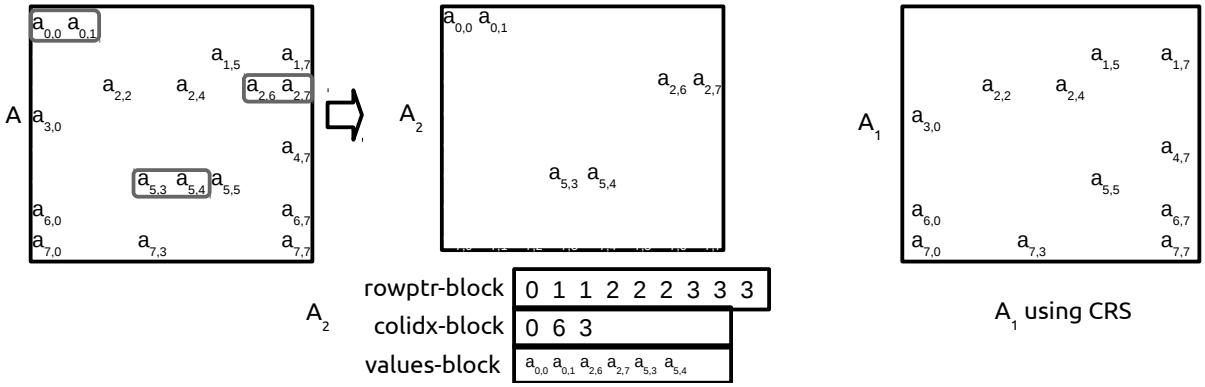


Figure 2.3: FSB matrix storage splits generate sub-matrices. In the presented example the matrix A is divided in two sparse matrices A^1 and A^2. Any sparse storage can be used of the resulting matrices but it is common to use a variant of the CRS storage that benefit of the contiguous values on the rows: $rowptr - blocks$ indexes block of values instead of scalars.

In the CRS format, contiguous values in a row give a good memory access, but no special optimization is made to process them whereas it can be an asset to unroll loops (prediction and pipeline of the instructions) and to have registers blocking. That is why, the FSB storage has been proposed in [21] and solves some of these drawbacks using a pre-processing of the matrix which can be costly but beneficial after one or maybe more SpMVs. The key idea is to extract contiguous block of values to process them differently and more efficiently. For example, all the blocks of α contiguous values inside a row from the original matrix A are stored in another matrix A_α. In the example shown in Figure 2.3, the matrix A is then equal to the sum of A_2, containing the blocks if tile 2, and A_1, containing the individual values. Instead of performing a single SpMV using A, we perform several SpMVs with the different block matrices. The storage of the obtained matrices can be the same, but we can imagine having different storage depending on the size of the blocks. In the original version, it was proposed to use a variant of the CRS and the size is given by $S_{FSB} = N_{NNZ} \times S_f + S_i \times \left(\sum_{b=1}^{MaxBlock} nb_{row} + \alpha_b\right)$ with $MaxBlock$ the size of the largest block and α_b the number of blocks of size b. The final memory occupancy can be greater than the CRS format depending on the quality of the blocking. The number of blocks and their sizes depend on the matrix itself but also on the permutations that have been applied to move the values contiguously. However, finding the ideal ordering which maximizes the number of contiguous NNZ (and minimizes the number of blocks) is a NP-hard problem. This storage was one of the first generic storages that tried to benefit of local NNZ patterns.

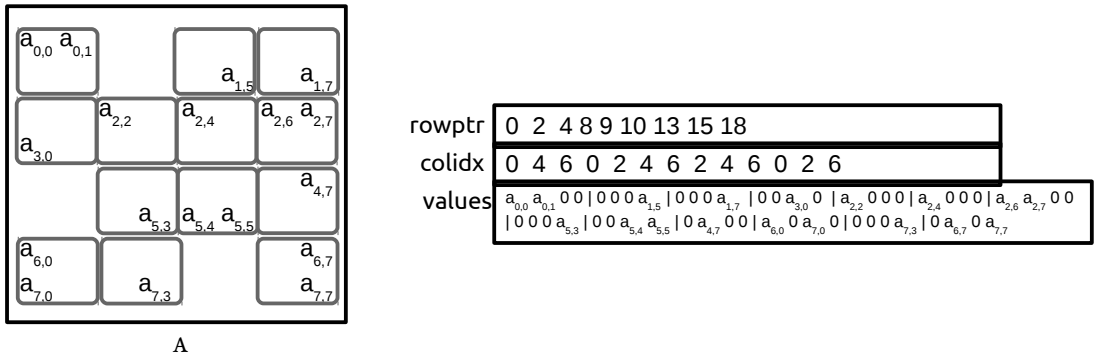


Figure 2.4: BCSR matrix storage example for the aligned version with a fixed block dimension of size 2×2.

Block Compressed Sparse Row Storage (BCSR)

In the FSB storage, the blocks are built from the concatenation of values from the same row while the BCSR extends this storage by blocking in the column's direction too. We refer to the dimension of the blocks by r and c for the number of rows and columns respectively. It has been originally proposed with blocks of size 2×2 in [22], but it has been extended with larger blocks of variable dimension in [18; 23]. Extracting the blocks from a matrix is complex and costly, especially with variable size, but it is then possible to use very efficient kernels. On the other hand, having a variable size of block reduces the zero filling which is potentially high when using a unique block size because a block of size $r \times c$ might contain a single NNZ. In the original BCRS, the blocks are assumed to be aligned: their upper left position is a multiple of the block dimension. Unaligned versions, like the unaligned block compressed sparse row (UBCSR), has been proposed in [24]. If we store the blocks and the position of their upper left corner (as a blocked COO) we obtain $S_{BCOO} = (r \times c \times \mathrm{B}) \times S_f + S_i \times \mathrm{B} \times 2$ with B the number of blocks. Otherwise, if we store them with a CRS approach, we have $S_{BCSR} = (r \times c \times \mathrm{B}) \times S_f + S_i \times (\mathrm{B} + m + 1)$. When the variable r is fixed and c is variable, the storage is called VBL (Variable Block Length) as shown in Figure 2.5. The size is given by $S_{VBL} = N_{NNZ} \times S_f + S_i \times (2 \times \mathrm{B} + m + 1)$ and if $r = 1$ there is no zero padding.

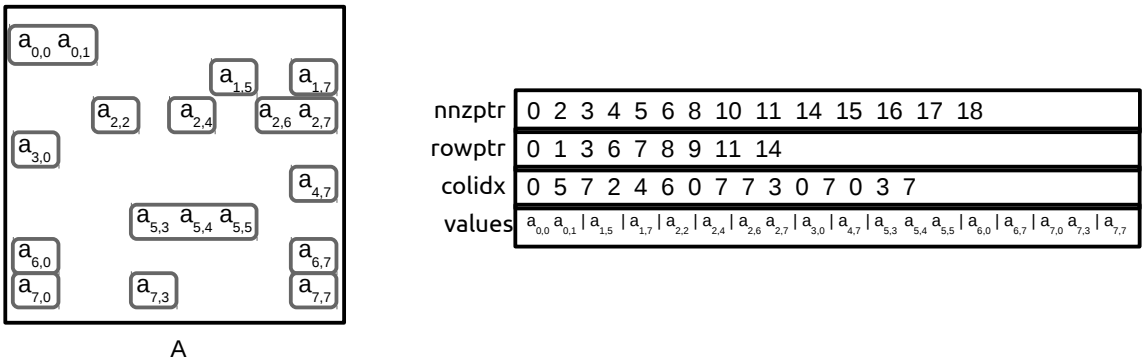


Figure 2.5: VBL matrix storage, with a block row dimension of $r = 1$.

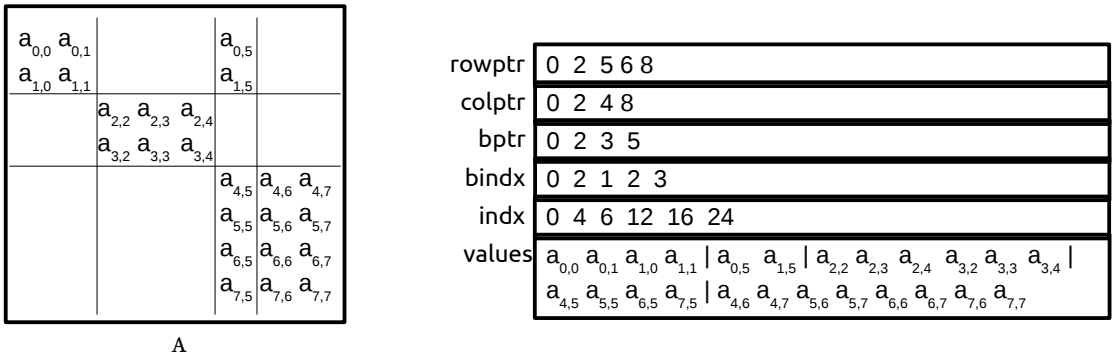


Figure 2.6: VBR matrix storage, a grid is built around the NNZ values.

Variable Blocking Row Storage (VBR)

The previous presented storages can be considered as generics, but it exists a class of storages that have been created for particular matrix shapes. The VBR storage is one of these specific storages, and it has been presented in [25]. This storage creates a grid around the NNZ values, and thus it is appropriate for matrices with large NNZ block areas. Even so, using this storage for a matrix with a different pattern is anti-productive and may increase the memory occupancy dramatically. As shown in Figure 2.6, the matrix is divided to create NNZ blocks without zeros and the borders of the blocks are valid for the entire matrix. In a *values* array, we store the NNZ in block order from top to bottom and left to right. The *rowptr* and *colptr* give us the grid by expressing where the matrix has been cut but without telling which blocks are filled. The *bptr* array expresses how many blocks there are on each grid line. The *bindx* tells in which grid column the blocks are located as it is done for the NNZ values in the CRS format. Finally, the *indx* array gives the positions of the blocks in the *values* array. The memory occupancy is $S_{VBR} = N_{NNZ} \times S_f + S_i \times (Grid_{col} + 1 + Grid_{row} + 1 + Grid_{row} + B + B + 1)$ with $Grid_{col}$ and $Grid_{row}$ the dimension of the grid and B the number of filled blocks.

Diagonal Storage (DIA)

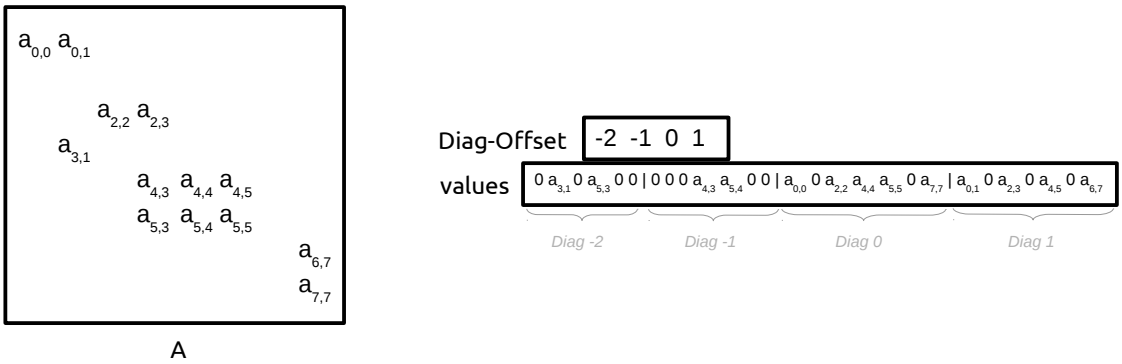


Figure 2.7: DIA matrix storage is optimized for matrix with values in the diagonals. In the presented DIA version, it is not possible to access directly a diagonal because the length of the diagonal is variable.

The DIA storage introduced in [25] is made for matrices that have values in a diagonal fashion

like the matrices from FEM for example. This storage keeps all the diagonals that contain at least one NNZ value: the diagonal $A(i+p, j+p)$ is stored if a NNZ is located at (i, j), for all p such that $0 \leq i+p < N$ and $0 \leq j+p < N$. If the NNZ are on the anti-diagonal $A(i, N-i)$ the complete matrix is stored. The values are stored in the *values* array in the diagonal direction, and the offset of each diagonal is written in $diag - offset$ as shown in Figure 2.7. The memory occupancy is given by $S_{DIA} = (\sum N_{Diag}(d)) \times S_f + S_i \times D$ with $N_{Diag}(d)$ the size of the diagonal d ($1 \leq N_{Diag}(d) \leq N$) and D the number of diagonals stored. The memory occupancy of this storage can be low because from the single offset index, we know the original position of the NNZ in the source matrix. Moreover, the computation of a diagonal can be optimized with register blocking, and the memory access pattern is regular.

Jagged Diagonal Storage Format (JAD)

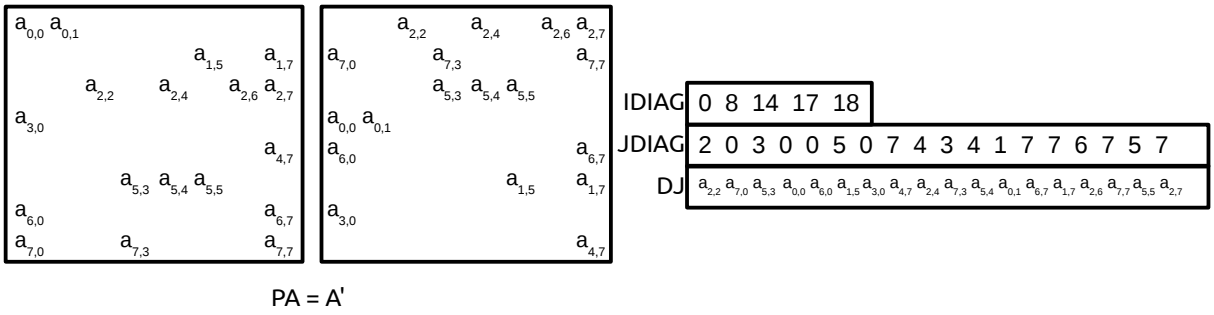


Figure 2.8: JAD

The JAD storage introduced in [26] is a good example of advanced structure to improve the memory occupancy and memory accesses in a sparse storage. As presented in Figure 2.8, the rows are first sorted according to the number and the position of their NNZ. After sorting, a row has at least the same number of NNZ than the one under it. The *values* array stores all the NNZ in column major: it contains all the first values of all rows, then the second values, etc. While the first row is always included, the number of rows that have a n^{th} NNZ decreases as n increase. This information is given by the *IDIAG* array which tells how many rows have at least n NNZ. As in the CRS storage, the *JDIAG* array contains the column indexes of the NNZ values. The memory occupancy is given by $S_{JAD} = S_f \times N_{NNZ} + (N_{NNZ} + N_{tjd} + 1) \times S_i$ where *Ntjd* here is the number of transposed jagged diagonals.

Automatic Kernel/Storage

The block based storages may have efficient kernels, but it has been shown in [27; 28] that there is not a perfect block dimension $r \times c$ because of the matrices and hardware variety. This is why some automatic methods have been developed to provide auto-tuning and optimizations and tried to find the more appropriate storage for a given context. These techniques remain generic and

thus we can expect to perform better when we work on a special shape with fined tuned kernel and storage.

2.1.2 Reordering

From the different storages presented, the ordering (permutations of the rows and columns) is important for the memory access pattern, the memory occupancy and the limitation of zeros padding. In fact, having blocks or any contiguous NNZ is clearly needed to have instruction pipelining and the use of SIMD, but a wrong ordering will fill the blocks with a lot of zero values, which may be anti-productive. General techniques which apply for a wide range of application have been proposed. A well known technique called Cuthill-McKee from [29] tries to make a matrix band-width by applying a breadth first algorithm on a graph which represent the matrix structure such that the resulting matrices have good properties for LU decomposition. However, the aim of this algorithm is not to improve the SpMV performance even so the generated matrices may have better data locality.

In [22] a method is proposed to specifically have more contiguous values in rows or columns. The idea is to create a graph from a matrix where each column is a vertex and by connecting all the vertices with weighted edges. The weights come from different formulations, but they represent the interest of putting two columns contiguously. Then we solve the Traveling Salesman Problem (TSP) to obtain a path that goes through all the nodes but only once and that minimize the cost of the total weight of the path. Therefore, a path in the graph represents a permutation; when we add a node to a path it means that we aggregate a column to a matrix in construction.

In the different score definitions, we use the following notations: n_j is the number of NNZ in the column j and $common(i,j)$ refers to the number of NNZ in common between columns i and j. The NNZ in common between two columns is the number of rows for which the two columns have a NNZ on as shown in Figure 2.9. From these definitions, if we have $n_i = n_j = common(i,j)$ then the two columns i and j have the same structure.

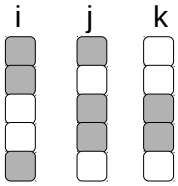


Figure 2.9: Values in common between columns of a sparse matrix of dimension $N = 5$ for two columns. The numbers of NNZ are $n_i = 3, n_j = 3$ and $n_k = 2$. The numbers of NNZ in common are $common(i,j) = 1, common(i,k) = 0$ and $common(j,k) = 2$.

The first formulation to obtain the weights of the edges is given by the following score

$$d(i,j) = common(i,j)\,. \tag{2.1}$$

This score is bounded between 0 and $Max(n_i, n_j)$ and the aim is to maximize the total score in a path. However, this score does not take into account the relative number of NNZ into the columns

and for example the same score is set if two columns containing each 1000 NNZ have 4 NNZ in commons or two columns of 4 values have all their values in common. Later on, more complex scores have been created to improve the accuracy and in [24] they propose the following one

$$d(i,j) = \frac{common(i,j)}{Max(n_i, n_j)}\,. \tag{2.2}$$

This score is bounded between 0 and 1 (achieved if two columns are similar) and the objective is to maximize the total score.

In [30] they proposed another formula

$$d(i,j) = n_i + n_j - 2 \times common(i,j)\,. \tag{2.3}$$

The score is between 0 (best) and $n_i + n_j$ when no values are common, therefore, the goal is here to minimize the total score.

The TSP is a NP-hard problem, and it is impossible to guarantee the obtention of an optimal solution. One of the methods to solve it uses a so-called greedy algorithm by generating a path using the best nearest neighbor algorithm. Then from a initial solution, which can be random, we can also better it using improvement heuristic as K-opt (usually 3-opt). However, in real application all the cost spends in the improvement of a matrix should be amortized by the gain in computation time, and some methods are not appropriate in real solvers but only interesting for the study/research.

SpMV with Space Filling Curve

We introduce the space filling curves (SFC) in Section 2.3.4 which are bijections from a n dimensional hypercube into a linear index: $\mathbb{R}^d \to \mathbb{R}$. While SFC are usually encountered when we work on spatial problems, it is possible to consider a matrix as a $2D$ map and to compute a SFC over it. In [31], they propose to use Morton indexing to reorder the value from a matrix and look at the improvement in terms of cache usage. In [32], they use Hilbert ordering with BCRS and show that it can be efficient for unstructured matrices. In [33], they use Hilbert ordering with a fractal storage and show interesting results in the case of multiple right-hand sides.

2.1.3 Parallel SpMV

SpMV on shared and distributed memory faces two mains problems. The first is how to divide the work without making the memory access worse than what it is in sequential, for example, by managing cache sharing. The second problem is how to balance the work and to avoid data replication everywhere.

One of the first great studies in multi-core SpMV has been presented in [34] where they use low level optimizations and study different hardwares. They conclude that having good performance depends on both the hardware and the matrix and that it is difficult to find general rules. In [35], the authors describe the problem of hierarchical memory across Simultaneous Multithreading (SMT). They show that it is needed to reorder the matrix but also to have an appropriate access pattern to

improve the performance. In [36], they concentrate on the distributed SpMV. Using a split method based on bi-partitioning in 2D, they try to minimize the communication volume. This is a difficult problem, and for example, they minimize the volume but not the number of messages. In [37], they introduce a new storage dedicated to the computation of Ax and A^Tx. They concentrate on multicore architectures, and their study is a good example of how the SpMV should be optimized for specific applications. In [38], they present a nice study by looking at various storages and architectures, and they remind how a static block size is important to have an efficient kernel but how it can have extra cost with a important zero fillings inside the block. They also show how difficult it is to find the good parameters.

2.1.4 SpMV on Accelerators

We summarize in the Appendix C.4 the hardware specificity of the GPU. Since the beginning of development of GPGPU (General-Purpose Processing on Graphics Processing Units) many researches have been done to improve the SpMV on such architecture [39; 40; 41; 42]. Some approaches use auto-tuning [43; 44] and it has been proposed to divide the input matrices between CPU and GPU, giving to GPU more appropriate parts (usually more dense parts). These studies show that the SpMV on GPU has a very low performance against the hardware capacity and motivates the use of blocking, which is crucial to improve the performances. Nevertheless, the result efficiencies are not greater than 5% of the peak performance either using CPU storages or GPU specific storages. In [45], they discuss about the difficulties of obtaining good performance on GPU and the need of investment to have a good kernel against CPU performance.The SpMV, as other irregular applications, is clearly not adapted for current GPU architecture and SDK. The optimizations of our implementation on GPU have been inspired by recent works that include efficient data structures, memory access pattern, global/shared/local memory usage and auto-tunning. The method to compute multiple small matrix/matrix products from [46] has many similarities with our implementation (e.g. the use of templates).

2.1.5 Experimental Examples

In order to illustrate the difficulties to achieve performances with SpVM, we present in Figure 2.10 some Flop-rates on modern CPU and GPU that we obtain for various matrix shapes and storages. We use vendor libraries Intel MKL [47] on CPU and CUDA sparse Blas [9] on GPU. We see that there is no ideal solution and that some storage perfom better for specific matrix shapes. The last plot of the graph presents performance for a small dense matrix computed several times; the idea is to hide the hierarchical memory access by having an important data reuse, and the performances are improved compared to other matrices, see Dense 200 (x 10000) in the figure. However, the performances are still low compared to the peak of the processing units which means that the memory is not the only constraint, and the underlaying kernels may not use register blocking or offer instruction pipelining. The DIA storage cannot be used for the Random Blocks 5/80000 since there is at least one value on most of the diagonal, this storage would require to allocate a

matrix of dimension 80000×80000. Moreover, the MKL DIA storage stores each diagonal in a vector of dimension N so in this case the total memory cost is $(2N - 1) \times N$. The results for the aligned block matrix are disappointing even for the BCSR. Finally, in the most difficult case when the values are randomly generated, the performances are clearly low, even so, many storages do not have zero padding.

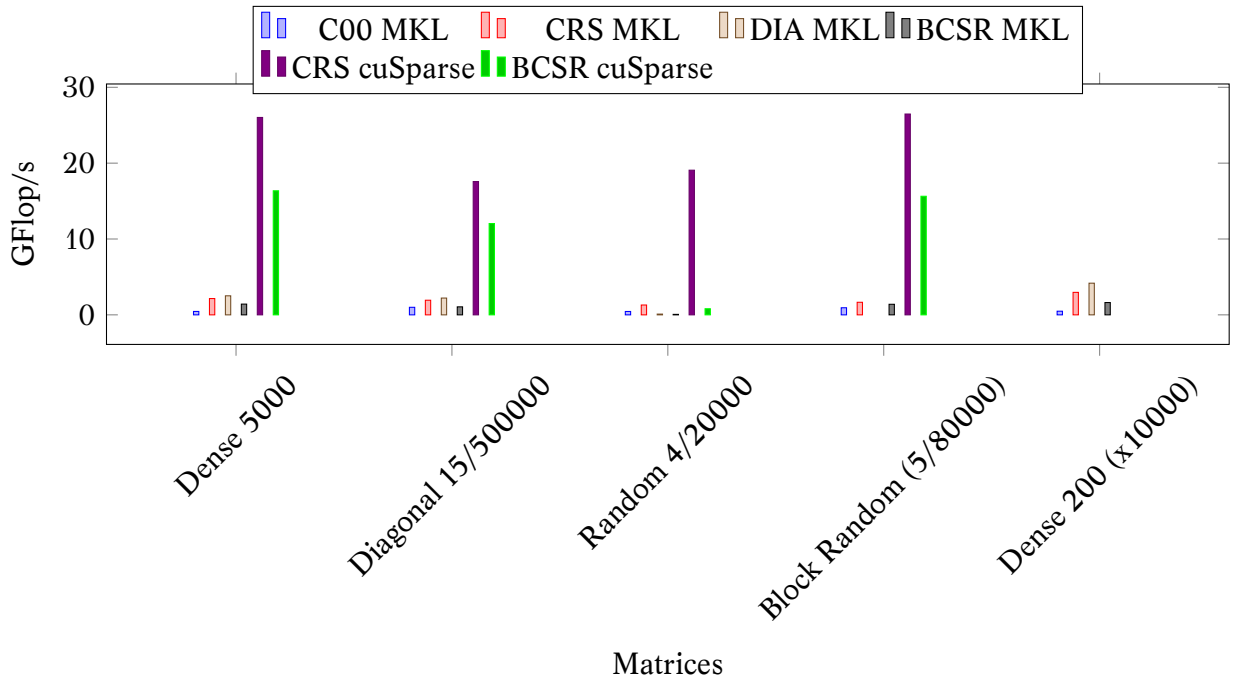


Figure 2.10: Example of SpMV using MKL and cuSparse for different matrices: dense, diagonal, random, random blocks of dimension 8×8 and the same small dense matrix computed several times. The format is p/N where p is the percentage of NNZ and N the dimension of the matrix. The block random cannot be stored using the DIA storage. The small dense matrix several times is not used on GPU since it is made to study the cache/data-reuse effect. The code is executed in sequential in double precision on a machine with a theoretical peak of $20 GFlop/s$ on CPU (Haswell Intel Xeon E5-2680 2,50 GHz) and $1.43 TFlop/s$ on GPU (K40-M). It has been compiled with Gcc 4.8.4, MKL 11.2 (2015.3.187) and Cuda 7.0 (7.0.28).

2.1.6 Summary

Different methods have been proposed by the recent researches to optimize the SpMV, but it has been shown that there are no perfect techniques, and that it depends on the architectures and the matrix shapes. Various performance models have also been proposed to help finding the best optimization parameters [48; 49]. In most cases, blocking clearly helps the locality, but it also involves filling with zero, and this can dramatically reduce the efficiency. Writing optimized code also helps with the use of SIMD intrinsics [50] for example. Reordering the matrix may be crucial but it can also be too expensive depending on the application; the cost of finding a good ordering and permuting the matrix must be covered by the gain in the SpMV. Finally, in general the SpMV hardly achieves 20% of the peak performance on CPU [34] and 5% on GPU [42].

2.2 Runtime Systems for Parallel Computing

In the field of HPC, a runtime system is in charge of the parallel execution of an application. A runtime system must provide facilities to split and to pipeline the work but also to use the hardware efficiently. In our case, we restrict this definition and remove the the thread libraries. We list here some of the well-known runtime systems: SMPSs [51], StarPU [52], PaRSEC [53], CnC [54], Quark [55], SuperMatrix [56] and OpenMP. These different runtime systems rely on several paradigms like the two well-known *fork-join* or *task-based* models. In the current study, we sometime use the term *tasks-and-dependencies* instead of *task-based* to explicitly point-out that we deal with tasks but also with the dependencies between them. We can describe the tasks that composed an application and their dependencies with a direct acyclic graph (DAG); the tasks are represented by nodes/vertices and their dependencies by edges. For example, the DAG given by $A \rightarrow B$ states that there are two tasks A and B, and that A must be finished to release B. Such a dependency happens if A modifies a value that will later be used by B or if A read a value that B will modify.

The *tasks-and-dependencies* paradigm has been studied by the dense linear algebra community [57; 58; 59; 60] and used in new production solvers such as Plasma [61], Magma [62] or Flame [63]. The robustness and high efficiency of these dense solvers have motivate the study of more irregular algorithms such as sparse linear solvers [64; 65] and now the fast multipole method.

2.2.1 Expression of the DAG from the Data-Flow

A tasks and dependencies DAG can be defined from a data-flow with implicit dependencies or a task-flow with explicit dependencies. In the task-flow, the user creates the tasks and manually expresses the dependencies between them as it is shown by Figure 2.11c for a simple example. This approach is expensive from the user point of view because he has to manage the dependencies manually, and the conversion of an existing application implies an expensive development.

On the other hand, in the data-flow approach, the runtime creates and manages the dependencies. The user inserts the tasks sequentially and tells how these ones access the data using the modes *read*, *read_write* and possibly *write*. Therefore, the runtime guarantees that the parallel execution respects this sequential insertion. The generated DAG using this model is always valid because the dependencies are generated from the sequential insertion, therefore, if a task A is inserted before a task B, it cannot generate a reverse dependency $B \rightarrow A$. Figure 2.11d shows an example of a DAG using this model and even if the implementations details are hidden the relation with the sequential code is straightforward.

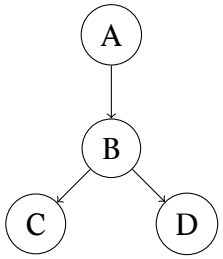
2.2.2 Other Data Access Modes

The basic modes, *read*, *write* and *read_write*, specify the data access of the tasks on the data, but it might be beneficial to provide more information. Therefore, several runtime systems provide advanced mode to define more accurately the data accesses and the relations between the tasks. As an example, StarPU proposes the *commutative* and *reduction* modes.

```
function A(in:d0, in:d1)
function B(in:d0, out:d1)
function C(in:d1)
function D(out:d0)

function Compute(d0,d1)
    A(d0, d1);
    B(d0, d1);
    C(d0, d1);
    D(d0);
```

(a) Sequential Code



(b) Corresponding DAG

```
T_A = create_task(A, d0, d1);
T_B = create_task(B, d0, d1);
add_dependency(T_A to T_B);
T_C = create_task(C, d1);
add_dependency(T_B to T_C);
T_D = create_task(D, d0);
add_dependency(T_B, T_D);
```

(c) Explicit-dependencies

```
create_task(A, READ, d0, READ, d1);
create_task(B, READ, d0, WRITE, d1);
create_task(C, READ, d1);
create_task(D, WRITE, d0);
```

(d) Implicit-dependencies

Figure 2.11: Explicit-dependencies vs. implicit-dependencies.

The *commutative* mode expresses the fact that some operations can be done in any order (but not at the same time). This mode might increase the parallelism of an application because many application have commutative operations, and it is difficult during the creation of the tasks to estimate in which order the data are going to be available. If a runtime does not support this mode, it can be replaced by a *write* where the computational order will be the same as the tasks insertion. We cannot represent with a DAG the fact that two tasks cannot be computed at the same time, even so we can add extra symbols to represent this relation.

The *commutative* mode may constrict the parallelism when the tasks depend on a few number of data because even if the tasks can be computed in any order, only one thread at a time accesses each data. Figure 2.12a illustrates this limitation with a simple example. StarPU proposes an additional access mode called *reduction* which is similar to the OpenMP *reduction* keyword in the loop and section statements. It expresses the fact that a data can be duplicated in order to have multiple threads that work with the copies and to merge the results into a final value. The *reduction* example in Figure 2.12b shows how the creation of a copy of A allows two threads to work concurrently. However, the *reduction* mode is expensive because it needs more memory to duplicate the data, and it also needs some CPU time to merge them into the final values. The real cost is implementation dependent because a *reduction* does not necessarily mean that a data is duplicated but that it can be duplicated if needed. In most algorithms, the *commute* and *reduction* modes are interchangeable.

2.2.3 StarPU

StarPU is a runtime system designed to manage heterogeneous architectures and with a implicit declaration of the dependencies based on a sequential data-flow. It shares several functionalities

```
task(commutative(A), Read(B), { A += B });
task(commutative(A), Read(C), { A += C });
task(commutative(A), Read(D), { A += D });
task(commutative(A), Read(E), { A += E });
Execution:
Thread 1 [
    { A += E; }
    { A += C; }
    { A += B; }
    { A += D; }
]
```

(a) 4 commutative operations

```
task(Redux(A), Read(B), { A += B });
task(Redux(A), Read(C), { A += C });
task(Redux(A), Read(D), { A += D });
task(Redux(A), Read(E), { A += E });
Execution:
Thread 1 [          Thread 2 [
    { A += B; }         { A' = 0; }
    { A += D; }         { A' += C; }
]                       { A' += E; }
                        { A += A'; }
                    ]
```

(b) 4 commutative/redux operations

Figure 2.12: Reduction Data Access Example.

with the OpenMP 4 standard but it also provides additional possibilities such as the customization of the scheduler. Moreover, in StarPU the main algorithm and the resulting DAG are de-correlate from the hardware and the different workers. The user creates the tasks without knowing where they are going to be computed but he gives different functions for the different processing unit types. Therefore, the choice is made at runtime by the scheduler that decides where a task will be computed. The scheduler is a critical component to achieve performance. This hardware abstraction from the algorithm definition is very important in the development of a long-term project.

By looking inside the StarPU implementation, we see that a StarPU scheduler is in charge of the management of the ready tasks and their assignment to the different workers. The management of the commutativity (using *commute* or *reduction* modes) is done by StarPU which performs early choices to decide which tasks to give to the scheduler. When several tasks share a commutative access to a data, only one of them should be marked as ready and given to the scheduler. Therefore, this gives to the scheduler a limited view on the existing tasks, and this makes the priority system secondary after the StarPU choices. If we insert the tasks A and B that both access the data k in a commutative mode and with the respective priorities *low* and *high*. StarPU has to choose one of the two tasks to be ready once k is available and gives it to the scheduler. Once the task is in the scheduler, its priority is taken into account to execute first the higher-priority task, but there is no guarantee that A will be ready before B. This is why it might be beneficial to replace some commutative data access by classic read-write to force the execution of some tasks before others even if it reduces the parallelism. In Appendix D.5 we present a scheduler that we developed to manage heterogeneous priorities.

StarPU-MPI

StarPU provides different methods to manage the distributed parallelization above distributed memory, and in this study, we use explicit calls to the StarPU-MPI *detached* routines *starpu-mpi-send* and *starpu-mpi-receive*. These so-called detached functions are managed like usual tasks in terms of dependencies: a *starpu-mpi-send* has a *read* access on the data and the *starpu-mpi-receive* a *write* access. However, they are not executed like user-level tasks because of the potential delay of the MPI communications. As an example, if a task calls a *mpi_send*, it can be extremely long before

the task finished, not because of the data transfer itself but because the addressee of the message must invoke *mpi_receive* to let the message be sent. Moreover, if the communications were inside normal tasks, we would have several threads calling MPI functions at the same time or calling asynchronous function without asking for a wait. To address this problem, StarPU creates an internal thread in charge of the MPI calls. When a task is over and relaxes an MPI detached function, the communication thread is responsible of the call. As a result, the MPI calls are performed as early as possible without involving any action from the user.

2.3 The Fast Multipole Method (FMM)

2.3.1 N-body Problems and Direct Computation

One refers to the problem of computing the pairwise interactions that N elements have together as a *N-body* problem. Such problems appear in a wide range of scientific fields, among which the most popular ones are astrophysics and molecular dynamics, namely the traditional particle-based simulations. Indeed, one of the first historical examples of this type of simulation, that is still studied nowadays, is the computation of gravitational interactions between astrophysical elements where one computes the interaction that each star/planet has with others and so for each of the N elements, we compute $N-1$ interactions, which leads to a $O(N^2)$ complexity. Let us consider for instance the smooth Laplace kernel used to compute the potentials and accelerations of particles from their position and mass/charge. The potential is given by

$$\varphi_i = \sum_{j=0, j\neq i}^{N-1} \frac{m_j}{r_{ij}} \,. \tag{2.4}$$

And the accelerations is given by

$$a_i = \nabla\varphi_i = -\sum_{j=0, j\neq i}^{N-1} \frac{m_j r_{ij}}{r_{ij}^3} \,. \tag{2.5}$$

Where the distance between element is

$$r_{ij} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2 + (z_i - z_j)^2 + \varepsilon} \,. \tag{2.6}$$

In case $\varepsilon = 0$, we need to ensure that we do not compute the interaction of a particle with itself, otherwise ε should be carefully chosen to be negligible. With a time integration, we obtain the acceleration and displacement and have a complete simulation. Computing the interactions between all particles is called *direct computation*, it is usually a compute bound operation (unlike the SpMV which is memory bound). In fact, in the Laplace example, we need to load $6N$ floating point numbers to perform between $20N^2$ and $40N^2$ *Flop*. This high ratio of operation *versus* data and the regular access pattern make the direct computation appropriate for GPU or the development of special hardware cards such as the GRAPE [66]. However, the quadratic complexity is a

clear limit to the computation of a problem of millions of unknowns in reasonable time. Various researches have been made to develop a new algorithm for the $n-body$ problem, for example, the cut-off method which does not compute the interactions between particles that are further than a given distance. Such methods might give accurate results when the decay of the kernel is high with the distance but it may not be accurate enough for kernel like $1/r$. Somewhere inbetween the cut-off methods and the full direct computation there exists hierarchical methods trying to efficiently approximate the far field while still using the direct computation between close elements.

2.3.2 Hierarchical Methods

In order to process differently the interactions depending on the distance, one needs a data structure that allows to retrieve this information quickly. One of the first published works about hierarchical methods for the $n-body$ problem was introduced in [67] for astrophysics. In this study, the authors construct a tree with particles at the leaves and internal nodes labeled with the centers of mass of their descendants. Then, depending on an accuracy parameter, they compute interactions between particles or between nodes. The underlying data structure is called a $k-d$ tree from [68] where k is the dimension of the space. Such tree is built by dividing the simulation box - a square in $2D$ or a box in $3D$ that includes all the elements of the problem - by two in each dimension successively as we add a level to the tree. The original algorithm has a $O(NlogN)$ complexity, but it was later reduced to a $O(N)$, see [69].

Later the Barnes & Hut method - from the names of the authors - has been proposed in [70]. This method has a $O(NlogN)$ complexity and introduced the octree in 3 dimensions (quadtree in 2 dimensions illustrated in Figure 2.13). Then using the same octree the fast multipole method (FMM) was proposed in [71] with a $O(N)$ complexity and an accuracy lower bound.

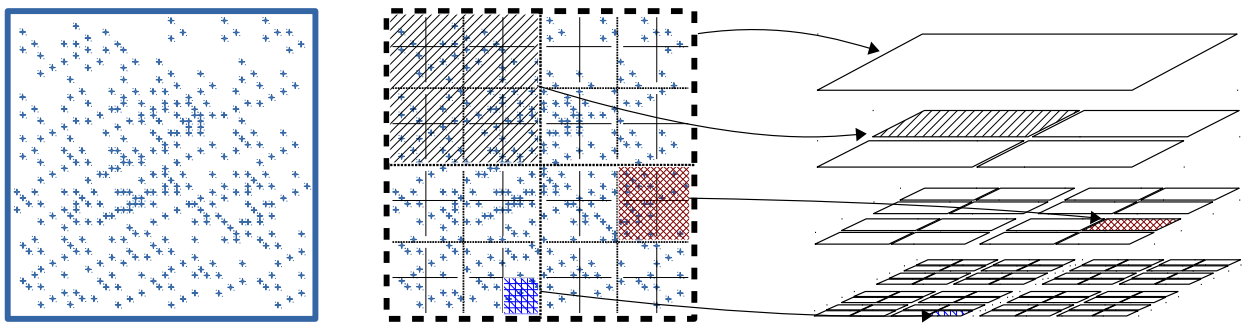


Figure 2.13: Quadtree (k-d tree of dimension 2), the space is divided by two in each dimension and to each division corresponds a cell in the tree.

2.3.3 The Fast Multipole Method (FMM)

The FMM is a hierarchical method for the $n-body$ problem introduced in [71] that has been classified to be one of the top ten algorithms of the 20th century by the SIAM [72]. In the original study, the FMM was presented to solve a $2D$ particle interaction problem, but it was later extended to $3D$. The FMM succeeds to dissociate the near and far field $\varphi = \varphi_{near} + \varphi_{Far}$, and still uses the

accurate direct computation for the near field. The original FMM proposal was based on a mathematical method to approximate the far field and using an algorithm based on a quadtree/octree for molecular dynamics or astrophysics. The algorithm is the core part because it is responsible for the calls to the mathematical functions in a correct order to approximate the interactions between clusters that represent far particles. When an application is said to be accelerated by the FMM it means the application uses the FMM algorithm but certainly with another mathematical kernel that matches the problems as we do for our TD-BEM accelerated by the FMM. The FMM is used to solve a variety of problems: astrophysical simulations, molecular dynamics, the boundary element method, radiosity in computer-graphics and dislocation dynamics among others.

The FMM Algorithm

One way to define the FMM algorithm is to consider that it answers the following question: *Having a mathematical method in hands which can aggregate the potential of particles from a box into a cluster and aggregate two clusters, and which is able to compute the interactions between these clusters and some particles, how can we reduce the complexity and ensure that all the particles interact together? In which order should we do the operations? One condition is added to ensure a minimum accuracy; if a cluster contains the potential of particles included in a box $B1$ of width $D1$ and is applied to particles included in a box $B2$ of width $D2$ then the distance between $B1$ and $B2$ must be greater or equal to $Max(D1, D2)$.* Figure 2.14 shows the type of operations that lead to a lower complexity.

Figure 2.14: The interactions between particles are replaced by interactions using clusters, thus leading to $N1 + N2$ interactions instead of $N1 \times N2$.

Similar to what has been proposed in the first hierarchical method, in the FMM a tree is constructed with the particles in its leaves (possibly several per leaf) and with each node representing their descendants. The nodes are composed by the multipole and the local parts: the multipoles represent the covered particles whereas the local part represent some interactions that will be applied to the covered particles. In the current study, we consider that the FMM root is located *at the top* and the leaves *at the bottom* such that the children of the cells from level l are at the lower level $l + 1$. In the first stage of the FMM the information from the particles is aggregated into the leaves, and this operation is called $P2M$ for particles to multipole. Then the information of the leaves is aggregated into their parents, and the operation is repeated from leaves up to the level 2 by the $M2M$ operation (multipole to multipole). After this upward pass, the nodes contain the informations of their descendants, and we need to apply them in a way to ensure all particles to receive the interaction from others. To do so, we compute the interaction list for all the nodes at all levels. The interaction list for a given cell c at level l is composed by the children of the neighbors

of c's parent that are not direct neighbors/adjacent to c. Then for each cell we compute the $M2L$, multipole to local, using its interaction list. After this transfer pass, all nodes contain the potential from the particles in a midrange distance. Then the downward pass applies the $L2L$, local to local, from level 2 to the leaf level by moving down. The $L2P$, local to particles, applies the far field to the particles. The direct interaction, between the leaves, is done at any time by the $P2P$ operator. All these operations are presented by Figure 2.15.

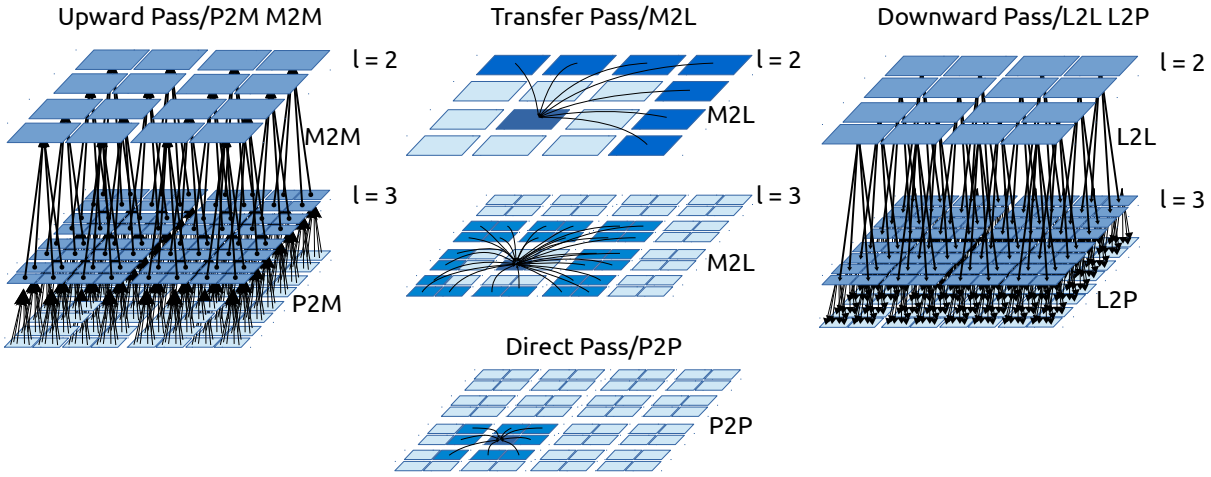


Figure 2.15: Fast multipole method algorithm

In Figure 2.16 we represent for a given cell how it receives the contributions from all the others. We see that depending on the distance, in space but also in the tree, the interactions happen at different levels. The same principle is applied to all nodes, which explains why the upward pass should first aggregate the contributions of the descendant.

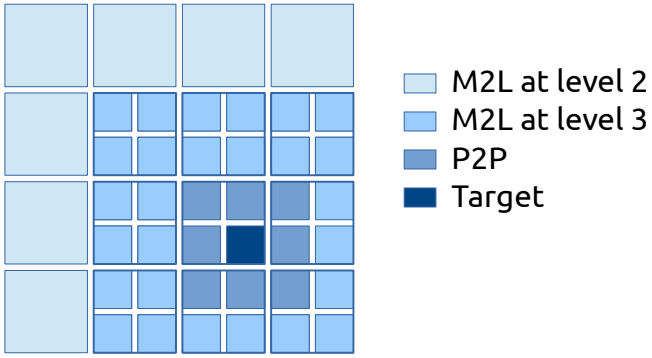
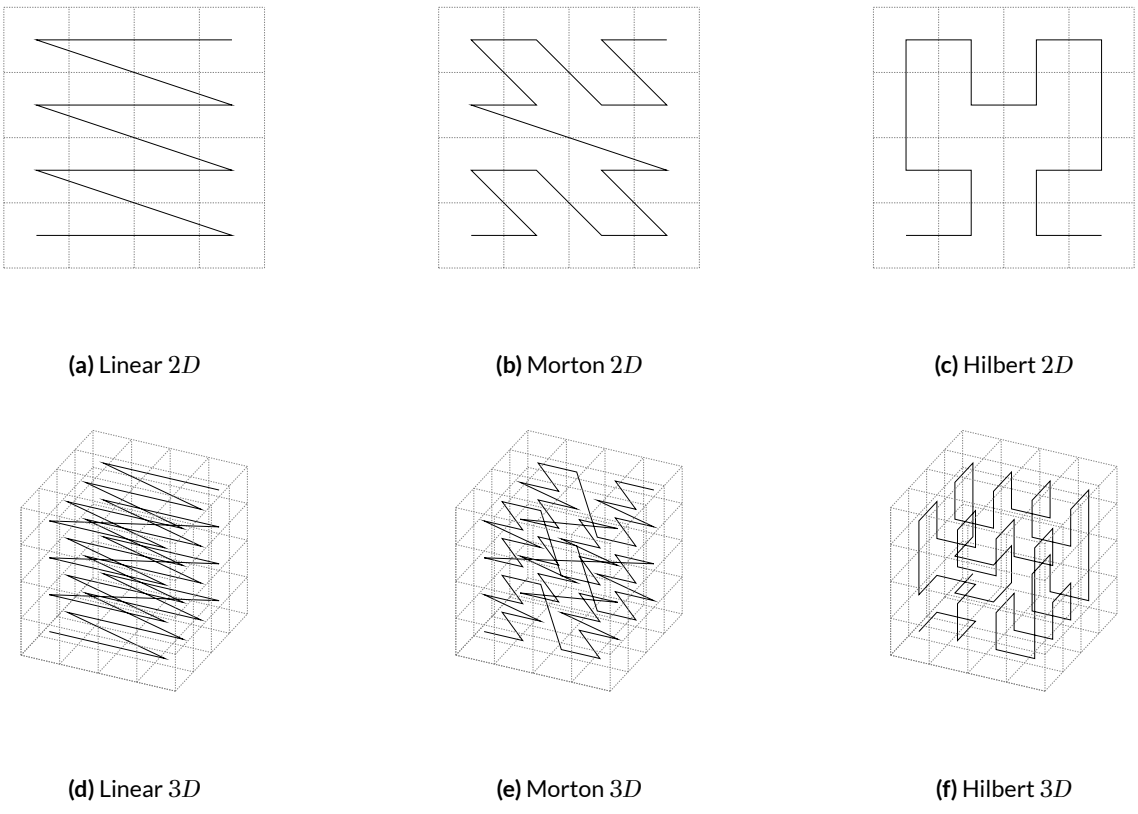


Figure 2.16: Complete interaction FMM for a given leaf

2.3.4 Space Filling Curves

In the FMM implementations, the way we access the cells in the octree and the way these are stored is crucial to have low complexity and good performance. In the case of a dense octree, when all the cells exist, for a dimension D and a tree height h the system is a uniform D-grid of dimension $g = 2^{h-1}$ in each direction and it leads to a total of $N^l = 2^{D\times(h-1)}$ leaves. One way to store them

is to use a dense multi-dimensional matrix. In this format, the index of a leaf is obtained from its tree coordinate $index = ((((d_1) \times g + d_2) \times g + d_3)...+)d_D$. Figure 2.17a illustrate this indexing in $2D$ and Figures 2.17d shows it in $3D$ where we have $index = ((x \times g + y) \times g + z)$. As it has been stated in the FMM algorithm, we need to access the neighbors of the cells: neighbors of degree 1 in the direct computation $P2P$ and from degree 2 to 3 in the $M2L$. Therefore, this linear indexing is not appropriate because close neighbors in space are always far in memory (except for the last dimension z). So when we access the neighbors of a cell, we may have a lot of cache misses and a poor data locality. Moreover, in distributed FMM if the cells are divided among the nodes following their indexes then the number of messages and their size may be extremely high. Better indexes have been proposed, and it is now common to use space filling curve (SFC) instead of linear indexing. In [73], they proposed to use the Morton indexing which is a SFC that was originally invented by Henri-Léon Lebesgue in $2D$ and then extended in [74]. It is usual to call this SFC the *Z*-curve from the pattern it creates in $2D$, see Figure 2.17b for the $2D$ and Figure 2.17e for the $3D$ examples. The second more used SFC in the FMM is the Hilbert [75] SFC which is usually called $U - curve$. Figure 2.17c and Figure 2.17f show examples of the Hilbert curve in $2D$ and $3D$ respectively.



(a) Linear $2D$ **(b)** Morton $2D$ **(c)** Hilbert $2D$

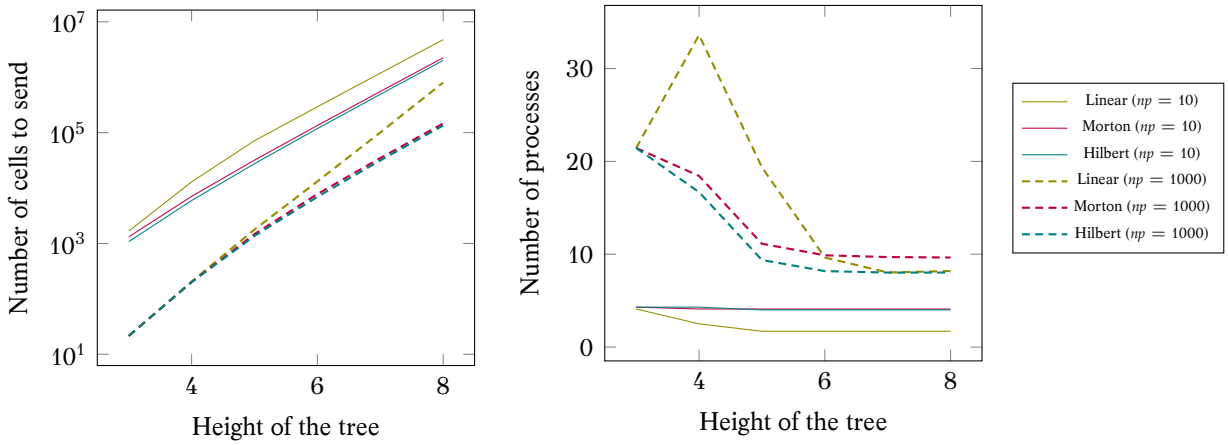
(d) Linear $3D$ **(e)** Morton $3D$ **(f)** Hilbert $3D$

Figure 2.17: Examples of three space filling curves.

The properties of these two curves have been widely studied, and it has been shown that even if the Hilbert curve has better properties in theory, in practice both are very similar [76; 77; 78; 79]. Moreover, the Morton index is very easy to compute in both directions compared to the Hilbert index and it is one of the reasons that make it much more popular. In appendix D.2 and ap-

pendix D.1, we give the algorithm to compute the Morton index and the Hilbert index respectively. For Morton, we also give the algorithm to compute the Morton indexes of the interaction list using a classical method, with the cost of converting back in the tree coordinate system, and using a bit-based method which is much more efficient, see Appendix D.2.1.

We have created two simple models to study the gain of using SFC compared to the linear indexing. In our first test, in Figure 2.18, we simulate a distributed memory FMM transfer pass at the leaf level. We consider that we have a dense octree and that each process has an interval of the full indexes: the process p is in charge of the leaves inside $[p * N^l/np; (p+1) * N^l/np)$, with np the total number of processes and N^l the total number of leaves which remains the same no matter the underlying index system. Then, we look at the communications by counting the number of cells each process has to send for a bottom transfer pass in Figure 2.18a. We also look at the number of processes that are involved in the communication in Figure 2.18b. For the number of communicating processes, the linear indexing performs well but when we look at the number of cells each process has to send, the linear indexing is much more expensive. This is easy to understand because the linear indexing divides the system in layers and even if each process shares its borders with a few others all its cells are potentially on the borders. On the other hand, we can see that the Morton and Hilbert indexes are very close.



(a) Average number of cells that each process has to send to others during the bottom transfer pass.

(b) Average number of processes that each process has to communicate with to perform exchange cells for the bottom transfer pass.

Figure 2.18: Space filling curve study for distributed parallelization: the figures show the communication properties when the tree is equally divided between the n processes.

In our second modeling presented in Figure 2.19, we estimate the number of cache misses when a single computational element performs the transfer pass at the leaf level. We consider that the computer uses a cache with a size C and that loading a cell of index i loads all the cell of index $page(i, C)$ to $page(i, C) + C$ with $page(i, C) = (i \, div \, C) \times C$. Again, the linear indexing appears clearly much more expensive once all the leaves do not fit in a single cache page. However, for the Morton and Hilbert indexes the results are very close, even so, the Hilbert index has slightly fewer caches misses.

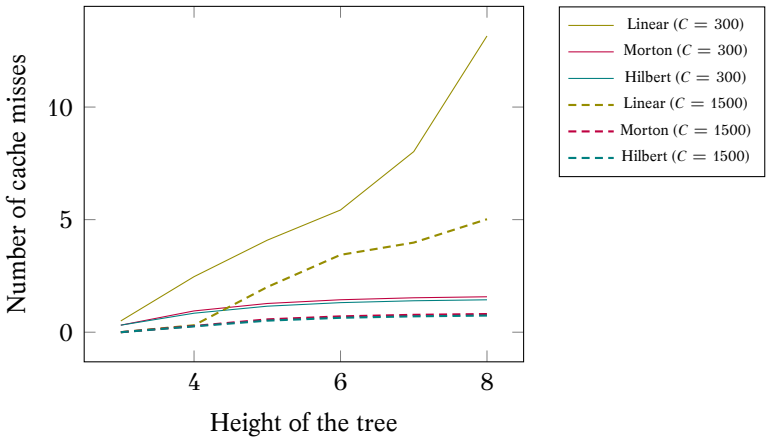


Figure 2.19: Space filling curve FMM cache Misses modeling : estimation of cache misses for different SFC and two cache sizes C during the bottom transfer pass. We consider that loading a cell load the C others cell that are in the same cache page.

Morton Indexing Examples

The FMM octree is built on top of a recursive subdivision of the space by two in each dimension: at each subdivision, the cells are divided into eight new cells in $3D$. At a level l the cell grid dimension is 2^l and the total number of cells $2^{l\times D}$. The linear indexing does the $\mathbb{R}^3 \to \mathbb{R}$ matching by *concatenating* the coordinate of the cell in binary

$$L(x,y,z,l) = ((x \times 2^{l*3} + y) \times 2^{l*3} + z = x_{l-1}...x_1x_0.y_{l-1}...y_1y_0.z_{l-1}...z_1z_0\,|_b\,. \tag{2.7}$$

The formula explains clearly the poor locality of this indexing because increasing the x coordinate by one increases the resulting index by $2^{l\times 3\times 2}$. Even so, this indexing has some advantages like its consistent pattern no matter the dimension of the problem.

The Morton indexing is intrinsically related to the hierarchical structure and the dimension of the problem. For a given cell c located at level l, the Morton indexing tells in which half of the subdivisions the cell is located starting from the root until level l. Indicating the correct half costs one bit and we need this indication for each level and each dimension, which gives us again the total number of cells $2^{l\times D}$. Finally, this structure can also be seen as an interleaving of the tree coordinate of the cells because the space is divided by two and we the use binary representation

$$M(x,y,z,l) = x_{l-1}y_{l-1}z_{l-1}...x_1y_1z_1.x_0y_0z_0\,|_b\,. \tag{2.8}$$

From this definition, a Morton index of a cell c is first composed by the Morton index of its parent, since they share the same path from the root, and then by D bits that tells where c is located among the last subdivision. Moreover, to find a cell in an octree, we need a Morton index and the level where the seek cell is located because different cells from different levels may have the same index: for example there is a cell with index 0 at every level. Figure 2.20 shows some Morton indexes in $2D$.

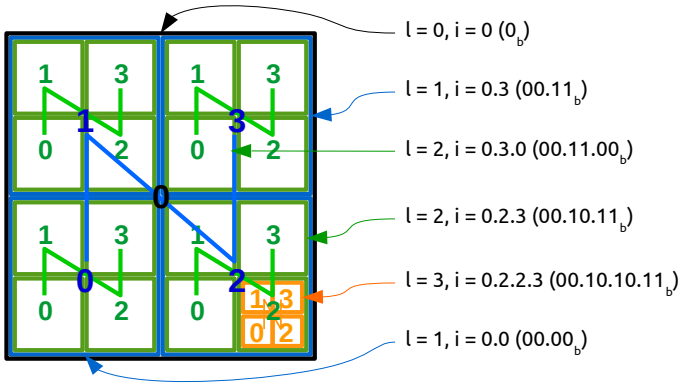


Figure 2.20: Morton index examples in $2D$. In order to retrieve a cell from an index the level is needed. The numbers X_b are the binary representation of the Morton indexes. The given example is in $2D$ and thus the bits are interleaved two by two: we need one bit to give the correct half in x and another for y.

2.3.5 ScalFMM

ScalFMM is an open-source software developed at the Inria Bordeaux which is dedicated to the HPC FMM. It was originally inspired by the FMB library [80]. It gives a particular attention to the genericity and the parallelization strategies using high level software engineering. The major points of ScalFMM is to provide facilities to develop new algorithms and kernels, or to execute an FMM with the existing modules. Its development has started before the presented work and is still ongoing. At the time of writing, ScalFMM is used in various applications: molecular dynamics, nanoscale physics [81], astrophysics, TD-BEM (current study), BEM and dislocation dynamics [82; 83].

ScalFMM octree data structure. The octree has to cover the simulation box and to include the particles or any other elements in its leaves. For a given height h, the number of cells in an octree is $2^{3\times(h-1)}$ in $3D$ such that it becomes impossible to allocate all the cells for a $h > 10$ in most machines. In addition, many applications have non uniform particle distributions, and in these cases, the cells that cover empty areas should not be allocated. Besides the memory occupancy, it is clear that the empty cells should not be included in the computation. For example, in the case of surface elements or in astrophysics where there is nothing between galaxies, there is more empty cells than occupied cells. Therefore, allocating a dense grid is impossible and building the octree with a link-list between parents and children may be inefficient in memory and will make the iteration and access to the cells costly.

In between a dense grid and a linked list, the authors in [80] propose a so-called tree by indirection. This data structure is similar to a virtual memory page table system where an index/address is composed of several sub-indexes to obtain the real piece of information. The tree is presented in Figure 2.21 and originally uses Morton index. This tree manages the sparsity of the system by allocating only parts of the complete octree depending on the existence of elements in the leaves. The tree by indirection has a parameter which is the height of the sub-octree h_s. Choosing a parameter h_s equals to 1 leads to a linked-list octree whereas having h_s equal to h is similar to a dense octree. When elements are inserted in the tree, we first allocate the sub-octrees from the root to

the host leaf if needed, and then we allocate the cells between the host leaf and the root. Our first ScalFMM version relies on an implementation of a tree by indirection with Morton index.

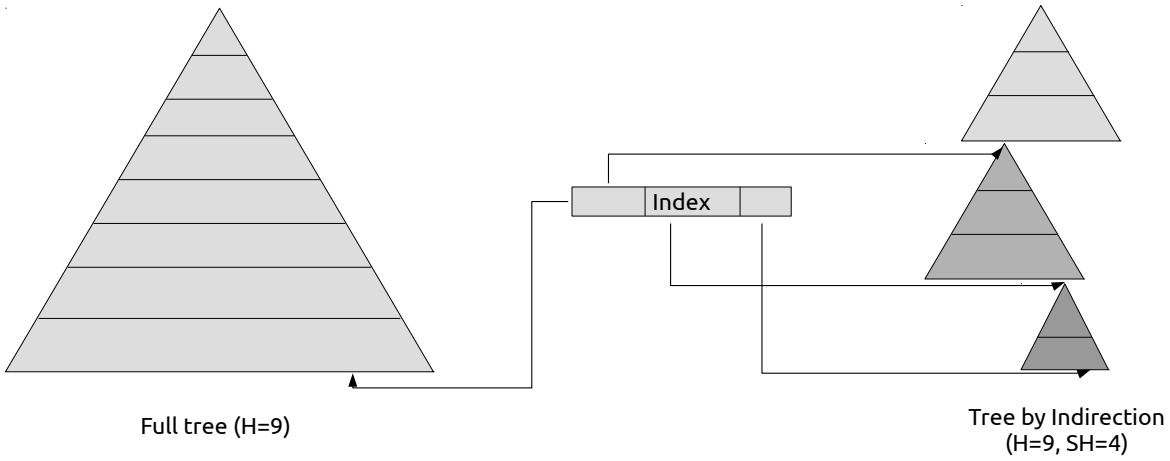


Figure 2.21: Tree by indirection. The index is composed of sub indexes to access the sub-trees.

ScalFMM kernels. ScalFMM comes with three kernels for the $1/r$ problems: Spherical Harmonics, Spherical Harmonics accelerated by rotations and Taylor expansions. It also provides new generation kernels that may be referred as black box kernels which are based on the Chebyshev or Lagrange polynomials [84; 85]. Each kernel has its own cell (Multipole/Local) but they can all be used with the different ScalFMM parallelization strategies. They have been developed by different researchers which in a way shows the advantages of ScalFMM.

Parallelization strategies. ScalFMM comes with several parallelized FMM algorithms using OpenMP on shared memory and hybrid MPI/OpenMP on distributed memory. On shared memory ScalFMM supports different paradigms: fork-join *for*, tasks-and-wait and section tasks-and-wait. Over the distributed memory, an extension of the hybrid MPI/OpenMP has been developed for the periodic case.

The MPI/OpenMP implementation has been developed before the current presented work but has been later improved and is presented in Section 4.3. The same has been done for the StarPU based version for which a draft was made earlier but rewritten and extended to distributed memory as presented in Section 4.4.

2.3.6 Other Parallel FMM

We state in previous sections that the FMM algorithm applies to various fields, and as result we cite only few of the numerous proficient implementations. The first parallel FMM applications were developed more than 20 years ago. In [73], the authors show result up-to 512 processors with an octree based on a hash-table. This study had proposed several schemes that are still used in modern parallel FMM as the indexing of the cells with Morton SFC. More recently, in [86], the authors describe a kernel independent FMM and are able to solve problem of 2.1 billion unknowns in a viscous flow simulation. Their application is parallelized with MPI and they show nice results thanks

to the dissociation of communications and computations. However, their Flop-rate and parallel efficiency are not as good as the latest researches.In [87], the authors describe a GPU based FMM for astrophysics and turbulence simulations. They prepare the data on the CPU and perform the computation on the GPU with an adapted data structure and by grouping the leaves that contain a small number of particles. In [88] the authors compare several FMM implementations for multicore and heterogeneous architectures. They show that the multicore implementations were still competitive if implemented carefully, but the GPUs have been improved a lot since. The authors from [89] provide a detailed study of the FMM on heterogeneous architectures. They describe the complete FMM steps from tree construction to computation and use up-to 30 nodes with 60 GPUs. Moreover, they give fine details about the theoretical and experimental cost of the different operations. In [90], they propose a pure accelerator-based FMM implementation and describe efficient CUDA kernels. The authors describe how the FMM operators can be efficiently implemented on GPUs. Large scale FMM has been proposed in [91] over $65K$ cores but also using CPU/GPU nodes. Lately, in [92], the authors introduced the PVFMM library which is an open-source FMM package ables to scale on thousands nodes and using accelerators (GPU or Xeon Phi). However, they do not rely on a runtime system and compute only the $P2P$ on accelerators which will limit the efficiency on non-optimal cases or for certain architectures. One of the earliest implementations of the FMM over a runtime system has been proposed in [93] for multicore architectures where the authors show a linear speedup up-to 16 cores.

2.4 Other TD-BEM Formulations for the Wave Equation

There are concurrent implementations and studies on TD-BEM solvers. In [94], the authors have implemented a TD-BEM application, and their formulation is similar to the one we use. They show results up to 48 CPUs and rely on the sparse matrix-vector product without giving details on the performance, and they do not provide up-to-date results in recent studies. In [95], the author uses either multi-GPUs or multi-CPUs parallelization and accelerates the TD-BEM by splitting near field and far field. The study shows an improvement of the GPUs against the CPUs and focuses on the formulation of the near/far computation. In [96], the authors give an overview of an accelerated TD-BEM for the wave equation by using the FMM. They argue to reduce the complexity to $O(N_S^{1+\delta} + N_T)$ (where $\delta = 1/3$ or $1/2$). However, their study does not focus on performance, but rather on a new formulation and its numerical study, which makes it difficult to compare to ours in terms of performance. The study from [97] presents a TD-BEM for the heat equation accelerated by the FMM with comprehensive details. Their kernel relies on the Chebyshev polynomial and they reduce the complexity from $O(N^2M^2)$ to $O(p^4q^2NM)$, where M is the number of time steps, N is the number of degrees of freedom p and q are the orders of the polynomial approximation in space and time.

2.5 Contributions

In the current thesis, we do not work on the TD-BEM formulation but rather on its efficient implementation and parallelization on modern HPC architectures. Moreover, we delegate to some black-boxes the construction of the NNZ values of the interaction matrices and the factorization of M^0, and we concentrate our work on the solution algorithm. The discretization of the mesh and the building of the input matrices are done by the original industrial application, and the solve of the linear system M^0 using a state-of-the-art library. The physical problems we simulate are based on rigid meshes (constant in time) and therefore, all the interaction matrices and the pre-computations needed by the linear solver - which includes analysis and factorization - are performed once at the beginning. Our contributions can be divided into three parts: the implementation of a parallel and efficient TD-BEM solver, the work on the parallelization of the general FMM algorithm and an attempt on the acceleration of the TD-BEM using the FMM.

From the original linear expression, we try to develop an efficient SpMV operator on CPU and GPU. On CPU, this work includes the permutation of the interaction matrices and the development of an unaligned block storage/kernel with SIMD instructions. On GPU, we define a new sparse matrix storage which is adapted to multi-streaming (or many-threads) processors. However, these kernels are still limited by the memory transfer and suffer of the zero padding. That is why, we propose to bypass the low performance of the SpMV by reordering the computation order and by using a custom multi-vectors/vector product. This part of the work is not an optimization or an improvement of the general SpMV because we use a custom operator who matches our needs and which is between the SpMV and the dense general matrix-matrix multiplication (GEMM). Nevertheless, the optimizations of our implementation have been inspired by the historical work on SpMV and some of our results are applicable to the SpMV. The interaction matrices have a sparsity pattern which is difficult to transform into dense blocks, but when we access to the NNZ values from a different manner we obtain matrices that have one dense vector per row. We study at different levels how this multi-vectors/vector product can be implemented on CPU and GPU to achieve high-performance. For the CPU, we develop a complex kernel which uses SIMD instructions and data re-use. The past researches of the SpMV on GPU show that data-blocking is mandatory to increase the Flop-rate. Therefore, we create two blocking schemes and their respective GPU kernels and study their efficiency and zero padding. The development of this complete matrix based solver includes efficient parallelization strategies over heterogeneous distributed nodes and a load balancing heuristic.

In our approach, we consider the FMM algorithm as a generic technique which is independent of the target application. We study the parallelization of the FMM using several paradigms in both shared and distributed memory. Starting from a sequential description, we explain how the FMM can be parallelized by classic fork-join methods and we incrementally move to a pure task-based FMM with the support of a runtime system. The *tasks-and-dependencies* expression of the FMM relaxes all the parallelism but the overhead of the runtime system in the management of the tasks/dependencies must be taken in account. That is why, we propose a new data structure

called *group-tree* to easily parametrize the granularity of the tasks and to target accelerators. We describe hybrid parallelization schemes using the classical pair MPI/OpenMP but also a task-based distributed FMM over StarPU-MPI. All these algorithms are included in the ScalFMM open-source library and available to the community.

The total execution time of a TD-BEM simulation is the summation of the times taken to generate the interaction matrices and the time to solve the linear system. We do not have the hand on the generation of the matrices but it is a costly step that can be even more expensive than the solve. Therefore, to avoid the generation of all the interaction matrices, it appears interesting to replace the accurate interactions based on these matrices by an approximation for the unknowns that are far from each other. This separation of the near and far fields can be done with the FMM and we implement the FMM kernel proposed in [3] for our TD-BEM. As a result, we have to generate the interaction matrices only between the leaves of the FMM tree which reduces drastically the complexity of this stage. However, the FMM solve seems expensive and complex to implement. Our kernel has been developed over the ScalFMM library which allows us to use its numerous parallelization strategies. Finally, we study some of the high level optimizations and provide preliminarily results, which call for numerous perspectives.

3

TD-BEM Matrix Approach

In this chapter, we describe the work around the direct/matrix computation to solve the TD-BEM. While the original linear system is expressed with SpMV, we have first worked on this standard operation and tried to optimize it. Then we have used another computational order and developed a new optimized kernel on CPU and GPU. This new order has lead to a new parallelization that we study experimentally on an airplane simulation.

3.1 An Attempt on the Optimization of the SpMV

The summation stage is the costly part and is naturally computed with SpMV when presented as in Equation 1.3, and thus our first work has been an attempt to increase the performance of this operator. In the background Section 2.1 we describe the state-of-the-art of SpMV and explain how difficult it is to achieve performance.

3.1.1 Matrix Shapes and Unknowns Ordering

In Section 1.2.2, we explain how the convolution matrices are generated and we point out the fact that there is a relation between the NNZ positions in the matrices, the spatial positions of the unknowns and the problem properties (time step, wavelength, etc.). If two unknowns are close in space, they will certainly impact others at the same time steps, which makes the matrices having two contiguous values in a row, and vice-versa which makes two contiguous values in a column. Therefore, to change the shape of the matrix, we need to perform some permutation of the rows/columns which means that we need to take the unknowns in a certain order. Attempting to improve the shape of a general matrix without knowing where its pattern comes from limits the possibilities, whereas here, we can take into account our knowledge of the matrices construction. The objective is then to improve the quality of the SpMV by padding with less zero, and having a better memory access pattern as stated in Section 2.1.2.

3.1.2 Reordering

In our current study, we describe two approaches that we used to improve the matrix quality; using the common elements graph (TSP) and using the Morton indexing.

Ordering based on Morton Space Filling Curve

In Section 3.1.1, we describe the relation between the ordering of the unknowns and the position of the NNZ in the matrices. Since we work on $3D$ mesh, one way to obtain a numbering of the unknowns is to use the space-filling curves (SFC) presented in 2.3.4. We compute the Morton index for each unknown by considering that a spatial box is over our mesh and with a high degree to guarantee that each Morton index covers very few unknown. Then, we sort the columns/rows using the Morton indexes and obtained a permuted matrix. This technique is easy to implement, and it gives a global ordering that can be used for all the convolution matrices, which means that from one SpMV to another, we do not have to permute the vectors. However, this technique is not $matrix-oriented$ and does not take into account the NNZ of the matrices, but we expect it to improve their quality from the matrix definition.

Common Elements Graph (TSP)

In Section 2.1.2, we introduce the TSP approach to permute a matrix that has been used in past research on the SpMV. We remind that the idea is to build a graph which expresses the interest of putting two rows/columns contiguously. The weights of the edges and the resulting score of the path that goes through all the nodes are based on the number of values in common between rows/columns: the number of common NNZ for two columns is the number of rows both columns have NNZ on. In Algorithm 1, we give a possible approach to build rapidly the table of elements in common between the columns of a matrix. Its complexity is $O(NNZ^2/dim)$ and thus it is usually much faster than comparing all columns pair by pair, which leads to a complexity of $O(dim^2 \times 2NNZ/dim)$. From the CSR storage, the algorithm iterates on each row and performs a double loops over the columns indexes to increase the common element table. Once we have the table of common elements, we can compute the score between columns. We have tested the three formulas presented in Section 2.1.2 (results are not shown in the present study) and obtained our best results with the score that we call Div^a_{max} given by

$$d(i,j) = \frac{common(i,j)}{Max(n_i, n_j)}. \tag{3.1}$$

We finally end up with a score table which tell us the cost of an edge in the graph for putting two columns contiguously.

Once the graph is built, we have to find the shortest path that goes on each node only once (TSP). This problem is NP-Hard, and it is impossible to guarantee the optimal solution for realistic problem size. One solution to get a path is to generate one using a greedy heuristic. It is also possible to improve an existing path using k-opt improvement method, but these approaches are

Algorithm 1: Finding the elements in common between columns. This algorithm has a time complexity of $O(NNZ^2/dim)$ and a spatial complexity $O(dim^2)$. From the output table, one should compute the final graph distance using for example Div^a_{max}.

```
Data:
function FindElementInCommon(integer csrRowCount[dim+1], integer csrColIdxs[NNZ],
dim, NNZ)
: Graph graph(dim, dim)
    Array elementsInCommon(dim, dim);
    for idxRow = 0 → dim-1 do
        for idxCol = csrRowCount[idxRow] → csrRowCount[idxRow+1]-1 do
            for idxColCommon = idxCol+1 → csrRowCount[idxRow+1]-1 do
                elementsInCommon(csrColIdxs[idxCol], csrColIdxs[idxColCommon]) += 1;
            end
        end
    end
    // elementsInCommon contains the number of elements in common between columns
    // And we can then compute the score in O(n^2)
```

expensive. Moreover, to study the SpMV we can use an expensive algorithm, but in a real solver the cost of the ordering, and thus the generation of the path, should be amortized by the gain in performance during the SpMV and so it cannot be too costly. A trivial greedy approach is to start from the closest nodes and then iterate by adding the next closest node to the path. We called this algorithm the $max - score$ heuristic, and it has a $O(N^2)$ complexity. We developed a new heuristic based on this $max - score$ that we called $max - block - score$ which tries to take into account the fact that we will block the matrix after the permutation. So a column should be added to a path not only if it has a good score with the latest node but also with the $c-1$ latest nodes (with c the number of columns in a block). This method, shown in Algorithm 2, has a complexity of $O((c-1)N^2)$, and it is an adaptation of the $max - score$ but should better match our objective. In this algorithm, we start by selecting the two nodes with the best scores which constitutes the initial path in the graph. Then, we look at the nodes that have not been inserted yet and add the best candidate in the left or the right extremities of the path. The algorithm stops once all the nodes have been proceed.

A more advanced heuristic has been tested by also taking into account the number of rows in a block r but it did not give any improvement compared to the presented methods.

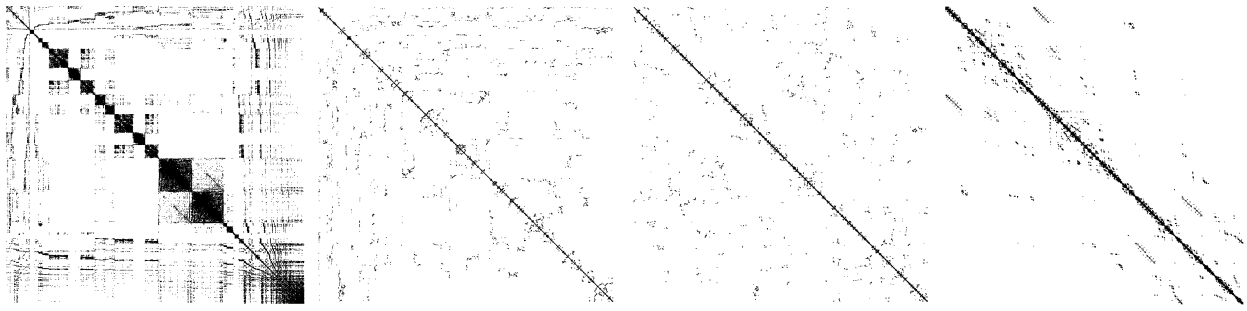
Experimental Quality Measure

We see the effect of a permutation by looking at the matrix shapes but also in our case by analyzing our mesh. That is why we present in Figure 3.1 the airplane test case M^0, describe in Section 1.2.4, for different permutations and in Figure 3.2 the same information but on the airplane mesh. This breaks the usual rule where we study a sparse matrix without taking into account its origin and the problem domain. In the matrix view, clearly there is an improvement when reordering with $max - block - score$, Figure 3.1b or with Morton, Figure 3.1d. In the airplane mesh, we see that the $max - block - score$ orders the unknowns that are spatially close, even so, it is independent from the mesh and has been applied on the matrix. It creates a kind of path on the mesh whereas the Morton method achieves this locality by the spatial recursive subdivision. Finally, when we apply

Algorithm 2: Greedy initial path construction - *max − block − score* - for the TSP problem. If the *BlockSize* variable is set to one, the algorithm turns to select the best candidate at each round like the *max − score*. The terms *left* and *right* illustrate the fact that we aggregate columns.

```
Data:
function GetGoodPath(Table scores(dim,dim), dim) : path[dim]
    path = ;
    // Get a first pair, like the best score in table for example
    [i:i'] = getBestScore(scores);
    // It is the starting point of the path
    path = {i, i'};
    // Iterate on remaining columns
    for FuncStyidxScore = 2 → dim-1 do
        [bestLeftScore, bestLeftNode] = inf , ;
        [bestRightScore, bestRightNode] = inf, ;
        forall the node ∉ path do
            leftScore = getScore(scores, path, left_dir, node);
            if leftScore is better than bestLeftScore then
                [bestLeftScore, bestLeftNode] = {bestScore, node};
            end
            righttScore = getScore(scores, path, right_dir, node);
            if righttScore is better than bestRightScore then
                [bestRightScore, bestRightNode] = {righttScore, node};
            end
        end
        if bestLeftScore is better than bestRightScore then
            path = {bestLeftNode, path};
        else
            path = {path, bestRightNode};
        end
    end
function getScore(Table scores, Path path, Direction direction, Node node) : score
    depth = Min(length(path), BlockSize);
    total = 0;
    for idxDepth = 0 → depth-1 do
        total += getScore(scores, getNthNode(path, direction, idxDepth), node);
    end
    return total;
```

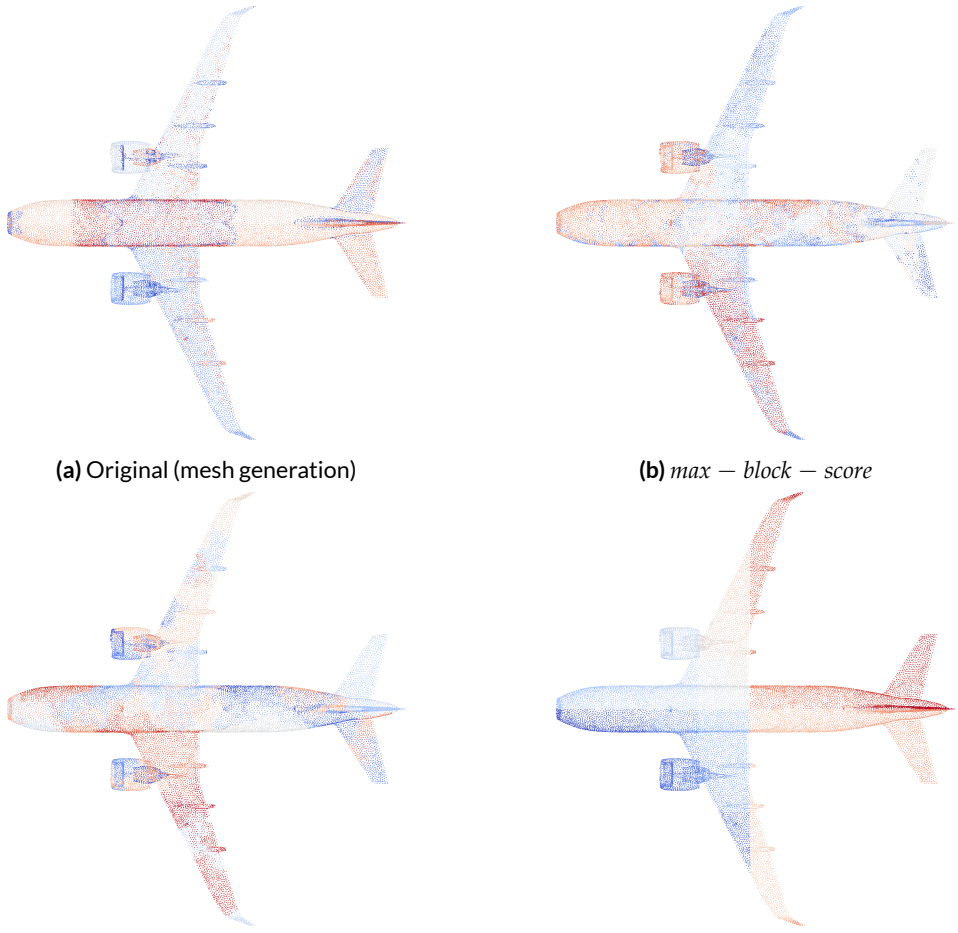
the 2-OPT heuristic to the *max − block − score*, we see that the path creates larger areas.



(a) Original (mesh generation) **(b)** $max - block - score$ **(c)** $max - block - score + 2Opt$ **(d)** Morton

Figure 3.1: Matrix view of the Airplane M^0 ordering. The black represents the NNZ or block of NNZ.

The past studies of the SpMV have shown that blocking is crucial, see Section 2.1. Therefore, the objective of our permutations is to improve the quality of the matrices for an unaligned block



(a) Original (mesh generation)

(b) $max - block - score$

(c) $max - block - score + 2Opt$

(d) Morton

Figure 3.2: Mesh view of the Airplane M^0 ordering. The path goes from red to blue: the resulting matrix is composed of the unknowns in red to the unknowns in blue.

storage; using an unaligned scheme reduces the zero padding in the blocks. Thus the best way to measure the quality is to count the number of unaligned blocks obtained from a permuted matrix. This is independent of the computational time, but we expect the computation time to be proportional to that number of blocks. Moreover, the higher the number of blocks the more important is the zero padding. We give in Table 3.1 the number of blocks we obtain for a small test case and in Table 3.2 for the airplane test case. We see that applying the $2 - OPT$ method gives an improvement in most of the cases, and for all the initial path but the number of blocks are just a slightly better compared to the greedy heuristics. For example, the $max - block - score$ approach is very efficient and only a few percent far from the best 2-OPT. In addition, it happens that we have more blocks once the 2-OPT has been applied like with the matrix $M69$ in Table 3.1, this is because while the path has been improved in terms of raw score, the generated matrix is finally not better. The generation of the unaligned blocks is done using the algorithm presented in Algorithm 3. However, here we show the ordering for four matrices as an illustration, but from our equation, we need to perform and SpMV for all matrices $M^{p>0}$ which makes the thing more complicated.

	$max - score$		$max - block$		Random		Default		Morton
M^k (NNZ)	Path	2OPT	Path	2OPT	Path	2OPT	Path	2OPT	
M^0 (18713)	2828	2735	2686	2611	13661	2710	3406	2781	2965
M^{23} (262172)	30904	30284	27196	27768	55272	27512	32678	30002	25826
M^{46} (60454)	4334	4364	4316	4412	26050	4722	4524	4338	5870
M^{69} (1014)	78	80	78	80	856	78	84	78	182

Table 3.1: Ordering quality for Div^a_{max} score and the cone-sphere C-927 test case introduced in Section 1.2.4. The table give the number of 4×4 blocks obtained from the unaligned conversion. For the graph based ordering, we give the number of blocks with and without 2OPT improvement.

	$max - score$		$max - block$		Random		Default		Morton
M^k (NNZ)	Path	2OPT	Path	2OPT	Path	2OPT	Path	2OPT	
M^0 (482078)	73806	69402	71912	69336	449067	69633	168279	69766	87898
M^3 (1570024)	185681	182121	181953	180194	1450148	182392	420448	181344	200594
M^6 (3256916)	347663	341925	337054	339817	2899655	343028	743984	342823	357188
M^9 (5207506)	540850	536092	525602	530406	4433638	537180	1103862	535184	555616

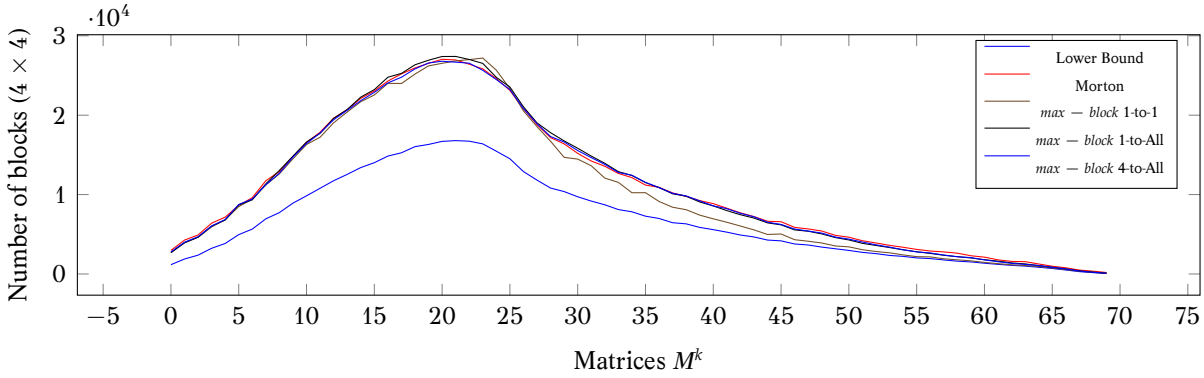
Table 3.2: Ordering quality for Div^a_{max} score and the airplane test case. The table give the number of 4×4 blocks obtained from the unaligned conversion. For the graph based ordering, we give the number of blocks with and without 2OPT improvement.

3.1.3 Reordering a Group of Matrices

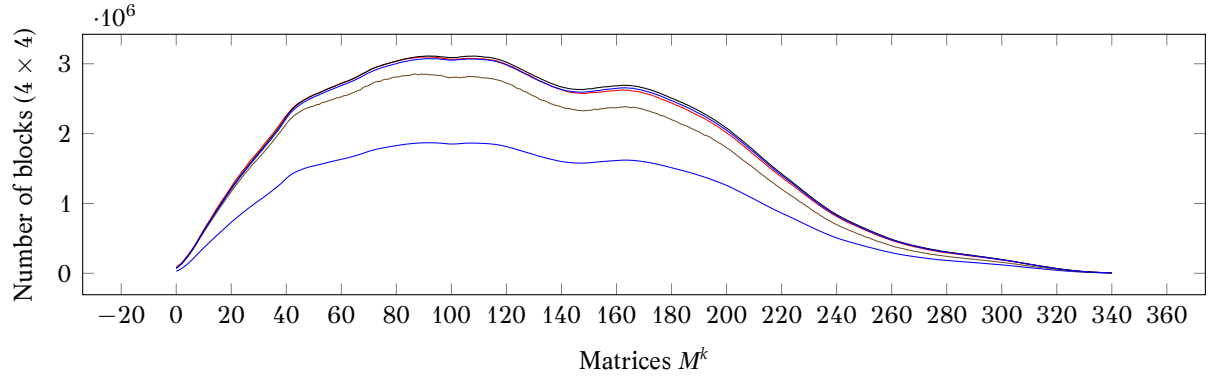
From Equation 1.3, we perform several SpMVs with the past states of the unknowns at each time step. If we try to find the best permutation for each matrix individualy/separately, we have to permute the past vectors to ensure coherency. To avoid permutating the vectors, we need to find a unique ordering for all the matrices, which is the case when using Morton indexing for example. The matrix M^0, in particular, represents the close interactions, and if we look at $M^{p<K^{max}}$, with $0 << p$ two columns will have a lot of elements in common if their respective unknowns are equally far from a important number of unknowns. But we can also expect this propriety to be true also for unknowns that are close because they illuminate the same unknowns and are illuminated by the same unknowns in return. That is why, we compare three quality of ordering variant: ordering each matrix individually ($1to1$), using the ordering from M^0 as a global ordering ($1toall$), using the ordering from x first matrices to generate a global ordering ($xtoall$). The Morton ordering is valid for all the matrices as a global ordering because it is based on mesh. With the $xtoall$, we cumulate the scores from the first x matrices before getting the ordering, because if we look at the common interactions for several time steps it should improve the future common interactions.

The Utopian number of blocks in a matrix is given per $LB = (NNZ + r \times c - 1)/(r \times c)$, while this number is certainly impossible to get, it gives a useful information to tell us how much zero padding we have. In Figure 3.3a and Figure 3.3b, for a cone-sphere test case and the airplane test case respectively, we look at the number of blocks we obtain for all the matrices from M^0 to $M^{K^{max}}$. In these figures, the minimal number of blocks possible LB is called $lower - bound$. As expected, when ordering each matrix separately we obtain the best results, but it has more than 30% zero padding and moreover, it would require to permute the vectors between the SpMVs and it is very expensive to generate all the orderings. The other methods are very close and lead to up to 50% of zero padding. It means that no matter how efficient is the SpMV used to compute these matrices,

it will waste 50% of the time computing with zeros. In results not shown here, we have also tested to keep ordering for a group of consecutive matrices, but it is difficult to tune such an approach.



(a) Number of blocks for the cone-sphere test case



(b) Number of blocks for the airplane test case

Figure 3.3: Number of unaligned blocks for all the matrices of two test cases and for different ordering strategies.

3.1.4 SpMV Implementations

3.1.5 Unaligned Block Coordinate Storage (UBCOO)

Once a matrix has been reordered, we propose a method, written in Algorithm 3, to quickly extract the unaligned blocks of static size $r \times c$. The algorithm generates the blocks in two similar passes with the first one needed to estimate the number of blocks before allocating them all. The memory is allocated in one shot and the second phase fills the block with the NNZ values. The time complexity is $O(NNZ \times c)$ and the space complexity $O(dim)$. We did not find a counterexample where the number of generated block is not optimal but we have not any proof that ensure the optimality. The optimal number of blocks is the minimum number of blocks to cover all the NNZ with potentially some zero padding (which is different from the lower bound).

3.1.6 CPU Implementation of UBCOO SpMV

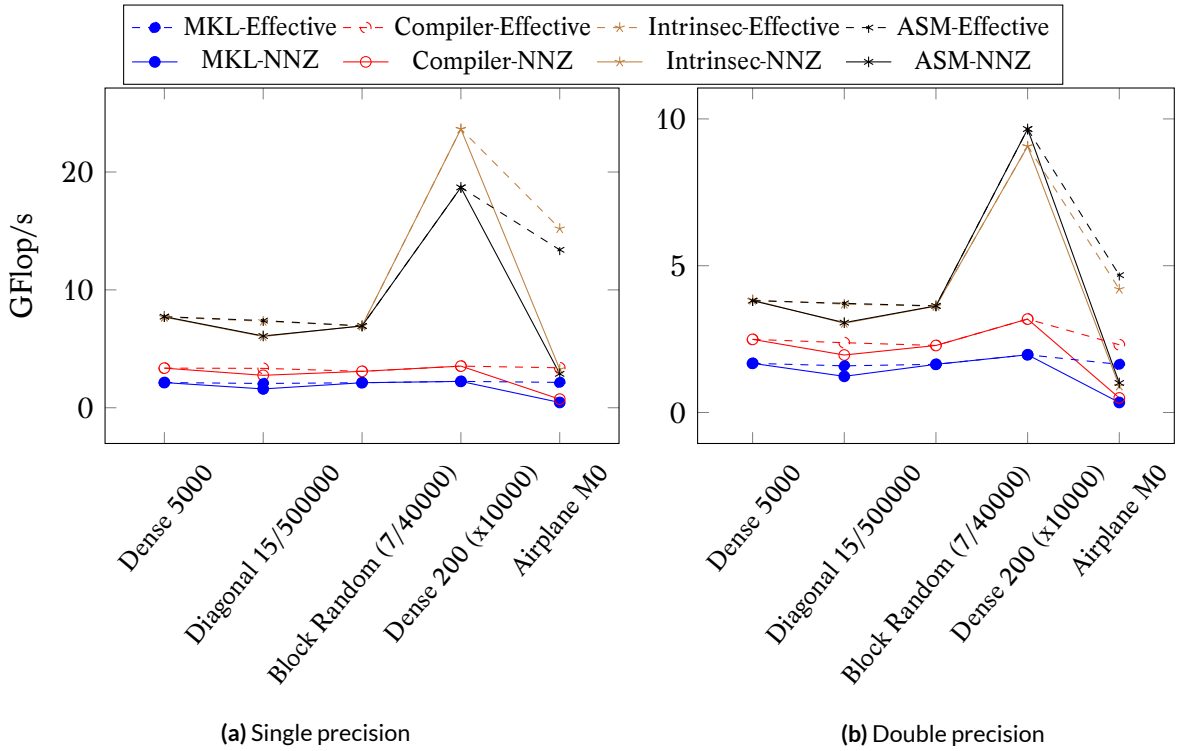
We describe in Appendix C.1 the key-points when developing an efficient kernel on CPU. We have implemented a SpMV kernel for our UBCOO storage with different levels of optimization. The basic version is developed in standard C with no special optimization except the ones from the compiler. We improve this version using AVX intrinsics in C ($C - AVX$) and then convert it to

Algorithm 3: Conversion from the coordinate storage (COO) to the unaligned block storage (UBCOO). The source storage can be changed to CSR storage without modifying the algorithm. The time complexity is $O(NNZ \times blockSize)$ and the space complexity $O(dim)$.

```
Data: Source NNZ should be accessed row by row
function CooToUBcsr(Coo cooValues[nbNnz], nbNnz, dim, blockSize)
: bcsrValues[nbBlocks × blockSize²], bcsrIndexes[2nbBlocks]
    integer rowMax[dim+blockSize-1] = -blockSize;
    integer nbBlocsk = 0;
    // First compute the number of blocks
    forall the NNZ value ∈ cooValues do
        if (value.i - rowMax[value.j]) >= blockSize then
            for idxCover = 0 → blockSize-1 do
                rowMax[value.j+idxCover] = value.i;
            end
            nbBlocsk += 1;
        end
    end
    // Allocate the arrays
    value − type bcsrValues[nbBlocks × blockSize²] = 0;
    integer bcsrIndexes[2nbBlocks] = 0;
    // Reset the working buffer
    rowMax[dim+blockSize-1] = -1;
    // Fill them
    integer idxBlock = 0;
    forall the NNZ value ∈ cooValues do
        integer currentBlock; if rowMax[value.j] == -1 OR (value.i - bcsrIndexes[rowMax[value.j]*2]) >= blockSize
        then
            // Start a new block
            bcsrIndexes[idxBlock*2] = value.i;
            bcsrIndexes[idxBlock*2+1] = value.j;
            for idxCover = 0 → blockSize-1 do
                rowMax[value.j+idxCover] = idxBlock;
            end
            currentBlock = idxBlock;
            idxBlock += 1;
        end
        else
            // Retrieve the block that cover the NNZ
            currentBlock = rowMax[value.j];
        end
        // Save the NNZ value
        integer rowOffset = value.i - bcsrIndexes[rowMax[value.j]*2];
        integer colOffset = value.j - bcsrIndexes[rowMax[value.j]*2+1];
        bcsrValues[currentBlock*blockSize*blockSize + colOffset*blockSize + rowOffset] = value.v;
    end
    // Here idxBlock == nbBlocks
```

hand-written assembly ($ASM - AVX$). These functions are presented in the appendix as code examples, see Section C.2 for the C-AVX and Section C.3 for the ASM-AVX. For these three different implementations, the block dimension is known at compile time. Of course, it is possible to compile several kernels for different block dimensions and choose the appropriate one at runtime. The performance results are shown in Figure 3.4 and we see that both AVX versions have similar performance. The pure *C* and the BCSR MKL versions are clearly not efficient. By looking more in

details details, we see that our *AVX* kernels have good instruction pipelining and that they use the registers correctly because they show good performance when computing the small dense matrix several times (which creates a memory reuse). However, even if the airplane M^0 matrix is showing good Flop-rate, the effective performance is quite low due to the zero padding.



(a) Single precision **(b)** Double precision

Figure 3.4: Results for the SpMV UBCOO 8×8 for different matrices: dense, diagonal, random blocks and several times the same small dense and M0. The code is executed in sequential on a Haswell Intel Xeon E5-2680 2, $50GHz$ with theoretical peaks of $20GFlop/s$ in single precision and $10GFlop/s$ in double precision. It has been compiled with Gcc 4.8.4 and MKL 11.2 (2015.3.187). The *NNZ* Flop-rate is obtain by not counting the zeros, and the *Effective* is the real number of operations that have been performed.

The ratio of effective/real Flop-rates is even worse for the MKL because it relies on aligned blocks. The number of aligned blocks needed to cover the NNZ of a matrix can be much higher compared to an unaligned scheme. For the airplane M^0 we obtain 35074 unaligned blocks of dimension 8×8 while the MKL uses 36158 aligned blocks of the same dimension. Finally, the MKL routine *mkl_dcsrbsr* used to convert the matrix into the the BCSR storage has been shown to be 2 to 10 times slower than our conversion routine, even so the construction of unaligned blocks should the more expensive. We can imagine that the MKL conversion routine is not using extra memory and is not allocating in one operation. But for both the computational kernel and the block conversion, it is difficult to understand this performance difference since the documentation is minimal and the source not available.

3.1.7 GPU Implementation of CBZ SpMV

We describe in Appendix C.4 the key-points when developing efficient kernel on CPU. We have extended our study with an attempt on NVidia GPU. We have implemented a GPU dedicated storage called Column Block Zone (CBZ) presented in Figure 3.5. The properties of the blocks

that are generated in the CBZ are based on the target GPU configurations: we need to know the maximum number of threads a group can have and the maximum shared memory per group. The maximum number of threads must take into account the GPU limit, and the number of registers used by our kernel against the number of registers for an entire group. From Figure 3.5, we see that we first divide the matrix by giving the same number of rows to each thread group. Then we create several blocks to cover all the columns: a block includes all the NNZ inside a column's interval that matches the size of the shared memory. Finally, the values from a block are stored in row major, and we use zero padding to ensure that all threads have the same number of values. This storage is not appropriate to CPU because of the zero padding, however, it may have some benefit of the data locality of the blocks.

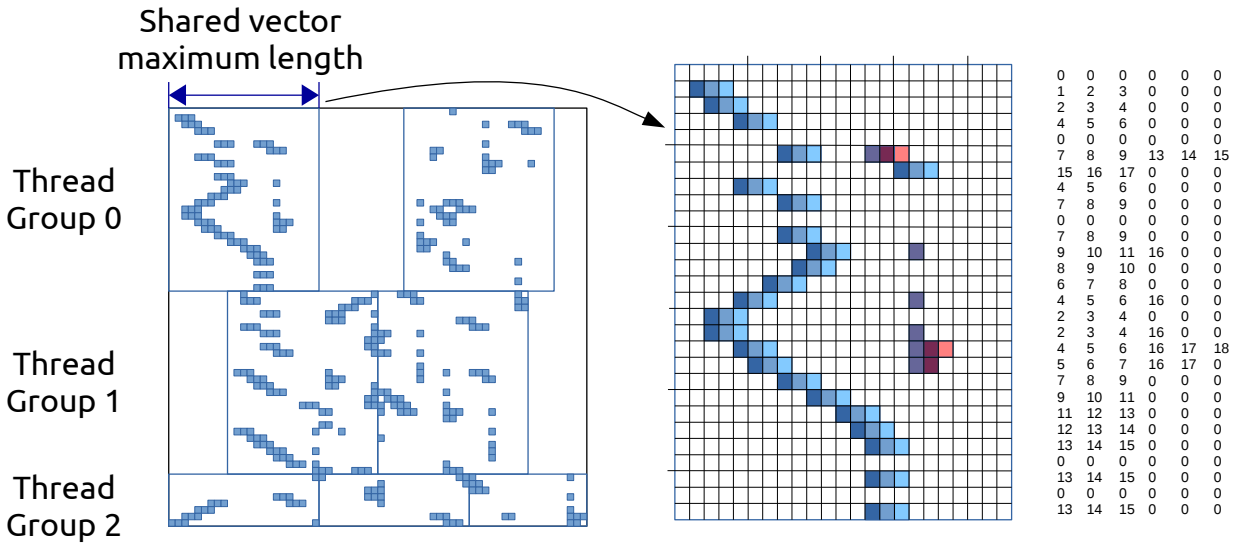


Figure 3.5: CBZ Storage for the GPU. The matrix is converted into block of dimension maximum threads per group $\times$ maximum shared memory per group. The NNZ are then stored in row major with zero padding to have the same number of values in each rows.

From the experimental results shown in 3.6 our kernel seems competitive against the CUDA cuSparse CSR. Even so, it is not appropriate when there is no large dense NNZ parts in the matrices, like for example in the block random (because the matrix dimension is large) and the airplane M^0.

3.1.8 SpMV Usage Summary

We have developed efficient algorithms to reorder the matrices based on the Div^a_{max} score or the Morton index and to generate unaligned blocks. These different stages of matrix preprocessing are expensive, and they are usable in a real application only if the SpMV is clearly taking advantage of them. Nevertheless, in our case, our optimized kernels while efficient in terms of raw Flop-rate are suffering of the zero padding. The final results are not sufficient and therefore the SpMV is certainly not the appropriate tool to improve our solver performance.

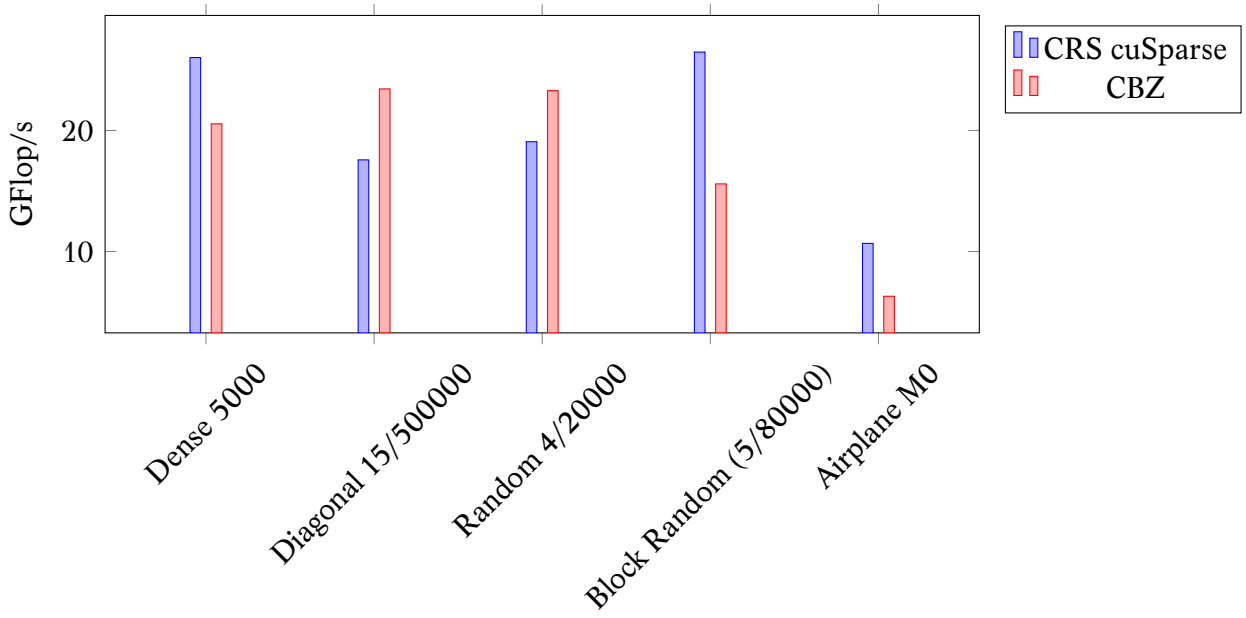


Figure 3.6: Example of CBZ SpMV on GPU different matrices: dense, diagonal, random, random blocks of dimension 8x8 and Airplane M0. The format is p/N where p is the percentage of NNZ and N the dimension of the matrix. The code is executed on a GPU (K40-M) with $1.43 TFlop/s$ of peak performance. It has been compiled with Gcc 4.8.4 and Cuda 7.0 (7.0.28). Our CBZ kernel is performing $19 GFlop/s$ for the Airplane M0 but the real Flop-rate is much lower.

3.2 Reordering the Summation Computation

3.2.1 Ordering Possibilities

A natural implementation of the summation stage is to perform K^{max} independent SpMVs using three nested loops. The first loop is controlled by index k in our formulation; it is over the interaction matrices, and it goes from 1 to K^{max}. The second and third loops are over the rows and the columns of the matrices and are indexed by i and j respectively. The indexes i and j cover the unknowns and go from 1 to N. The complete equation is written in Equation (3.2) where all indexes n, k, i and j are visible.

$$s^n(i) = \sum_{k=1}^{K_{max}} \sum_{j=1}^{N} M^k(i,j) \times a^{n-k}(j) \,, 1 \leq i \leq N \,. \tag{3.2}$$

In terms of algorithm, it is not mandatory to keep the outer loop on index k and two other orders of summation are possible using i or j. The three possibilities are represented in Figure 3.7 where all interaction matrices M^k are shown one behind another and represented as a $3D$ block. This figure illustrates the three different ways to access the interaction matrices according to the outer loop index. The natural approach using k is called by *front* and usually relies on SpMV (Figure 3.7a). We decide to use the approach called by *slice* using j as the outer loop index (Figure 3.7c) because the resulting matrices have a particular structure which is appropriate for optimizations.

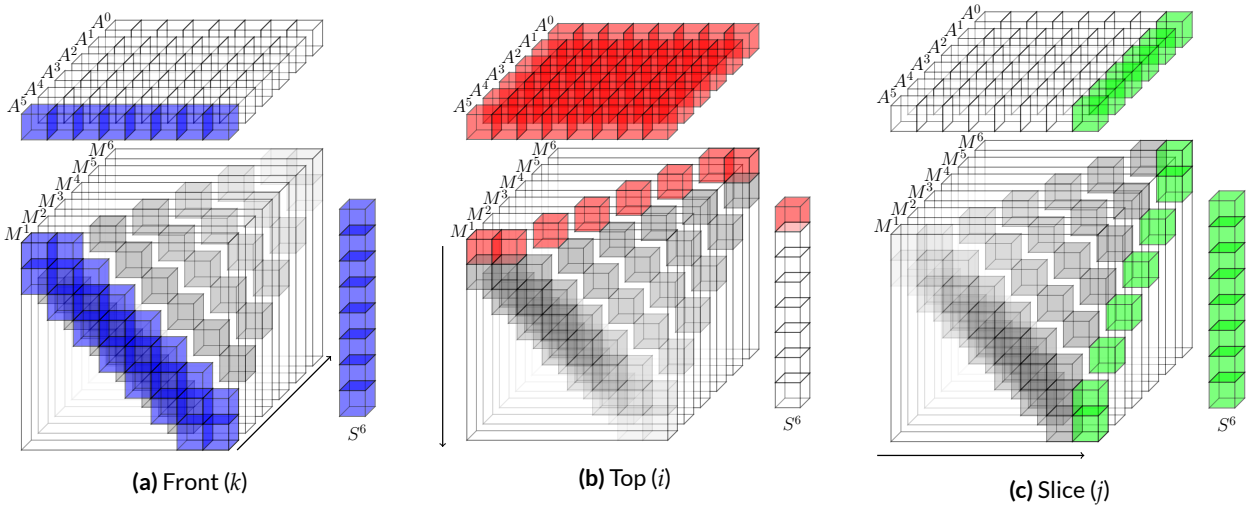


Figure 3.7: Three ways to reorder the computation of s^n with current time step $n = 6$, number of unknowns $N = 8$ and $K^{max} = 6$. (a) The outer loop is in the different M^k matrices. (b) The outer loop is over the row index of M^k and s^n. (c) The outer loop is over the column index of M^k.

3.2.2 Slice Properties

We denote by *slice* the data when the outer loop index of the summation is j in Equation (3.2). A $Slice^j$ is composed by the concatenation of each column j of the interaction matrices $[M^1(*,j)\, M^2(*,j) \dots M^{K^{max}}(*,j)]$ as illustrated in Figure 3.7c. This definition induced the relation $M^k(i,j) = Slice^j(i,k)$ and thus a slice is a sparse matrix of dimension $(N \times K^{max})$. It has a non-zero value at line i and column k if $d(i,j) \approx k \cdot c \cdot \Delta t$, where $d(i,j)$ is the distance between the unknowns i and j. While an interaction matrix M^k represents the interaction between the unknowns for a given time/distance with coefficient k, a $Slice^j$ represents the interaction that one unknown j has with all others over the time. This provides the main property of the sparse structure of a slice: **the non-zero values are contiguous on each line**. As illustrated by Figure 1.4, it takes several iterations for the wave emitted by an unknown to cross over another. In other words, for a given row i and column j all the interaction matrices M^k that have a non-zero value at this position are consecutive according to index k. In the slice format, it means that each slice has one vector of NNZ per line, but each of this vector may start at a different column k which correspond to the delay between the emission and the reception of a wave [*]. If it takes p time steps for the wave from j to cross over i, then $Slice^j(i,k) = M^k(i,j) \neq 0$ for $k_s \leq k \leq k_s + p$ where $k_s = d(i,j)/(c\Delta t)$. We refer to these dense vectors in each row of a slice as the row-vectors. Using the interaction matrices, we multiply a matrix M^k by the values of the unknown at time $n-k$ (i.e. $a^{n-k}(*)$) to obtain s^n. By working with slices, we multiply each $Slice^j$ by the past value of the unknown j (i.e. $a^{*<n}(j)$). An example of a slice is presented in Figure 3.8.

When computing the summation vector s^n, we can perform one scalar product per row-vector. Then, s^n can be obtained with $N \times N$ scalar products (there are N slices and N rows per slice) instead of K^{max} SpMVs.

[*] This might not be true with some specific features such as different propagation media, a ground plane, symmetries or periodicity. In most of those cases, there more than one NNZ vector per line but the presented study is still valid.

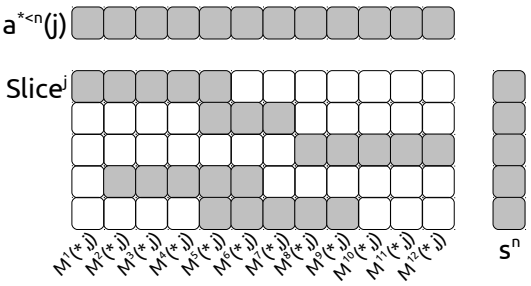


Figure 3.8: An example of *Slice* where the non-zero values are symbolized by gray squares and the zero values by white squares. The vector $a^{*<n}(j)$ contains the past values of the unknown j. The vector s^n will contain the current result of the summation stage for $t = n$.

3.2.3 Slice Computational Algorithm with Multiple Steps

The scalar product is a level 1 BLAS, and it has a low ratio of Flop against loaded data. In fact, one needs to load one value from both vectors to perform one multiplication and one addition. More precisely, by calling d the dimension of the vector, we need to load $2d + 1$ values to perform $2d$ Flops. Figure 3.9a shows how we compute a slice using one scalar product per row. We propose two optimizations to increase this ratio.

The first optimization is to work with multiple summation vectors at a time. At time step n, we compute the current time step summation vector s^n and the next step summation vector s^{n+1} together. When computing a summation vector, we use the slice matrices which remain constant and the past values of the unknowns relatively to the target vector time step. The vector s^n requires the past values a^p, with $n - K^{max} \leq p < n$, whereas the vector s^{n+1} needs the past values $a^{p'}$ with $n - K^{max} + 1 \leq p' < n + 1$. In other words, s^{n+1} needs most of the values that are used to compute s^n but it also needs the current state a^n which has not been computed yet. If we replace a^n by zero, we are able to compute s^n and a part of s^{n+1} together but s^{n+1} is incomplete. By doing so, we perform a matrix-vector product instead of a scalar product. The vectors are the non-zero row-vectors of the slices, and the matrices are the past values which match the summation vectors s. We denote by n_g the number of summation vectors that are grouped together and Figure 3.9b shows the different computation possibilities with $n_g = 3$. Once the summation is done, if $n_g = 2$, we end up with s^n and s^{n+1} and we continue the usual algorithm for s^n to obtain a^n after the resolution step (Equation (1.5)). Then, this result a^n is projected to s^{n+1}, using a single SpMV and M^1, and allows us to obtain the complete summation vector s^{n+1}. We refer to this projection as the radiation stage. It is possible to have n_g greater than 2, but the higher n_g, the more important the radiation stage. In this configuration, we load $d + n_g + d \times n_g$ data to perform $n_g \times 2d$ Flops.

The second optimization takes into account the properties of the past values. When working with the $Slice^j$, we need the past values of the unknown j: s^n needs $a^{n-K^{max} \leq p < n}(j)$ and s^{n+1} needs $a^{n-K^{max}+1 \leq p' < n+1}(j)$. The vector $a^{n-K^{max}+1 \leq p' < n+1}(j)$ is equal to the vector $a^{n-K^{max} \leq p < n}(j)$ where all values are shifted by one position, and with the first value equals to $a^n(j)$ (or zero if this value does not exist at that time). In order to increase the data reuse, we consider only one past value vector for the n_g summations involved in the process. We take the existing values $a^{n-K^{max} \leq p < n}(j)$ and append one zero to each $n_g > 1$ as it is shown in Figure 3.9c. In this case, we load $d + n_g + (d + n_g - 1)$

data to perform $n_g \times 2d$ Flops. This operator is called multi-vectors/vector product and is detailed in Figure 3.10.

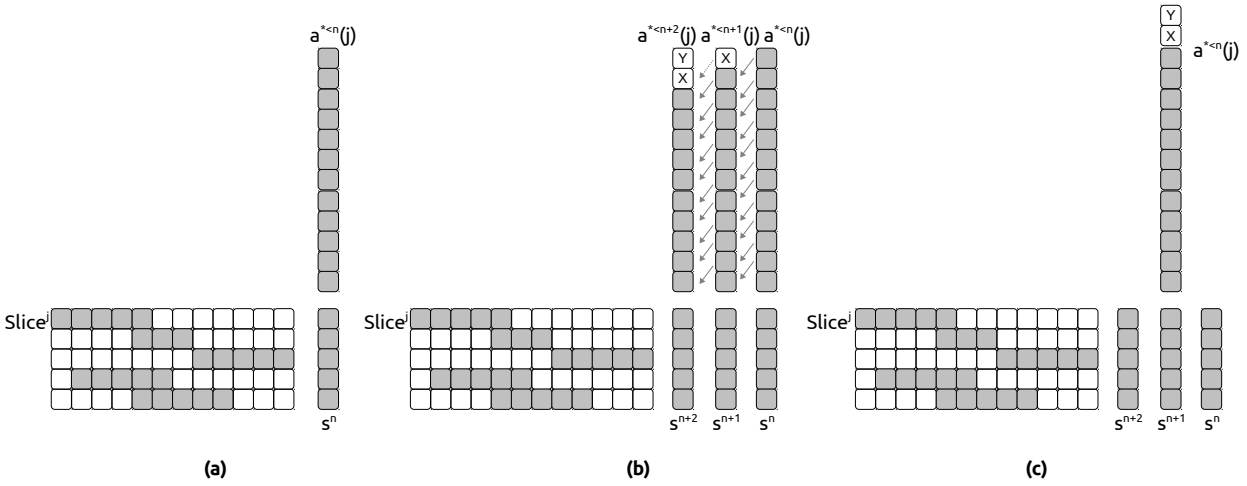


Figure 3.9: Summation stage with $Slice^j$ and 3 possibilities. (a) With $n_g = 1$ using scalar product. (b) With $n_g = 3$ using matrix/vector product (X and Y are the values of a, not yet available and replaced by zero). (c) With $n_g = 3$ using the multi-vectors/vector product.

The computation of the multi-vectors/vector product can be done in many ways where the more naive implementation is to perform a scalar product for each of the n_g results. With a such approach, we delegate the data reuse to the hardware cache hoping that, since all the scalar products involved use almost the same values, the performance will increase. However, it is also possible to implement this operator with more advanced algorithms as shown in Section 3.3.

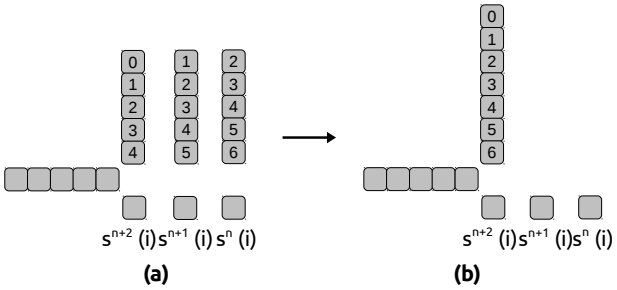


Figure 3.10: Computing one slice-row with 3 vectors ($n_g = 3$): (a) using 3 scalar products and (b) using the multi-vectors/vector product.

3.3 Multi-Vectors/Vector Product on CPU

The multi-vectors/vector product is parametrized by two sizes v the length of the row-vector and n_g the number of solutions. The number of solutions n_g should be carefully chosen to have enough data reuse but limited radiation cost. To introduce an optimized implementation of this operator, we can look at the number of uses for each value. The values from the row-vector are used n_g times: one time per resulting vector. The values from the past vector are used n_g times, except for the values at the beginning and the end: the first value is used once, the second twice, and so on until the n_g^{th} value which is used n_g times.

One possible implementation is oriented by limiting the number of memory global reads (in the sense of vector accesses). For each values from the row-vector that is read, we can apply it to the n_g result vectors; with this approach each row-vector value is read once and thus it costs v distinct loads (we cannot do better). For the past vector values, we propose to use an intermediate array which contains the past values that matches the current computing position in the slice; each past value is read once and copied into the intermediate array and later used for the n_g result vectors. For the past vector the number of loads from the memory is $v + n_g - 1$ but the number of loads from the intermediate array is $n_g \times v$ for the entire multi-vectors/vector product. Our intermediate array is of dimension $n_g - 1$, and we perform a shift operation of its values at each iteration to avoid reading from the past vector. This method is presented in Algorithm 4. We describe in Section 3.3.1 how this algorithm can be implemented with SIMD and in Section 3.3.2 how we can improve its performance.

Algorithm 4: Multi vectors/vector product

```
Data: n_g the number of result vectors to compute simultaneously (should be ≥ 2)
function MultiVectorsVector(vec[SIZE_VEC], past[SIZE_VEC + n_g - 1]) : res[n_g]
    register res[n_g] = 0;
    // We store the first past values (to load them once)
    register buffer[n_g-1];
    for idxBuffer = 0 → n_g-2 do
        buffer[idxBuffer] = load(past[idxBuffer]);
    end
    // For all values in the vec
    for idxVec = 0 → SIZE_VEC-1 do
        // Copy the current vec value
        register value = load(vec[idxVec]);
        for idxRes = 0 → n_g-3 do
            res[idxRes] += value * buffer[idxRes];
            // Shift the buffer value for the next idxVec loop
            buffer[idxRes] = buffer[idxRes+1];
        end
        res[n_g-2] += value * buffer[n_g-2];
        // Load a new value from the past vector
        buffer[n_g-2] = load(past[idxVec+n_g]);
        res[n_g-1] += value * buffer[n_g-2];
    end
    return res ;
```

3.3.1 SIMD Multi-Vectors/Vector Product

We give a description of the Single Instruction Multiple Data (SIMD) in Appendix C.2, and we remind here that SIMD are instructions which perform the same operation on multiple values that are contiguous in memory. In the original multi-vectors/vector, from Algorithm 4, we use scalar floating point value. One can delegate the optimizations to the compiler and hope that it will be able to use SIMD instructions but in such an approach, the compiler may try to replace scalar mathematical operators by SIMD instructions. Whereas, with the usage of SIMD intrinsics, the algorithm can be improved in a more complex way.

Instead of computing a scalar, the outer loop can be unrolled by the number of values in the SIMD data type l_{SIMD}. This conversion continues by transforming the intermediate array into SIMD data type, and thus shifting its values is similar to a shift of size l_{SIMD}. Figure 3.11 shows an example of the SIMD algorithm for a data type size $l_{SIMD} = 2$. To ensure performance, the length of the row-vector v should be a multiple of l_{SIMD} otherwise extra tests are required. This condition is possible by padding with zero which in the worse case may add $N \times N \times (l_{SIMD} - 1)$ zeros.

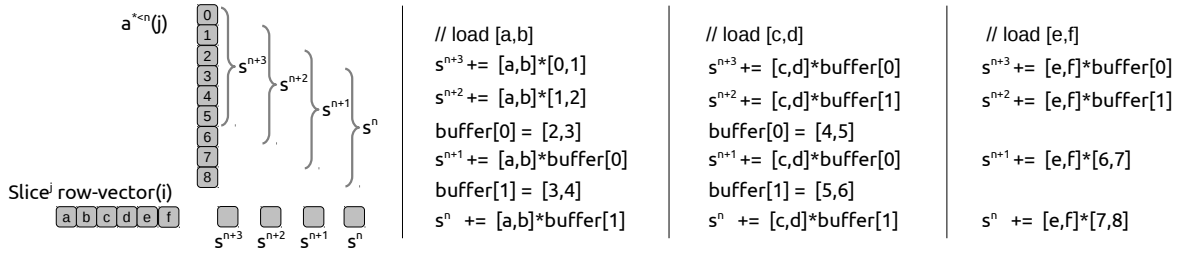


Figure 3.11: SIMD Multi vectors/vector product, with $n_g = 4, v = 6$ and SIMD data type size $l_{SIMD} = 2$.

The complete algorithm of this operation is given in Algorithm 5. Using *C++* template, described in Appendix E.1.1, this algorithm can be used with any n_g size and any SIMD types (as SSE or AVX for instance). Depending on the parameters n_g and l_{SIMD}, the compiler will remove some loops because they will never be used.

3.3.2 Memory and Assembly Optimizations

In the presented multi-vectors/vector algorithms, we load the past data into an intermediate array because it has many benefits in terms of memory access optimizations. First, the memory from the past vector may not be correctly aligned while it is an important criterion for the load SIMD performance, and that is why by moving the data into a local array we can ensure aligned accesses. Secondly, we avoid aliasing because we work with local data (there are no multiple pointers possible to our local array). Finally, we limit the cache pollution because we read from the global arrays only once, and this makes the LRU cache management appropriate: each value is read once and never used again. However, another improvement is to use the CPU registers to store the working variables instead of the memory (and the cache). It is possible to advise the compiler to use a register for a given variable using the *register* key-word but this is not guaranteed. Using the registers is very efficient because they are the fastest memory, and many instructions require at least one operand to be in a register. But for large n_g, the available registers on a CPU may not be sufficient to avoid the usage of the RAM. Finally on previous generation architecture, the usage of an intermediate array instead of several variables - which is possible only if loops are completely unrolled - was less efficient.

In assembly we can play directly with the registers. As a result, the intermediate array of the previous algorithm is no longer an array but composed by several registers. Moreover, the n_g intermediate results for the current computed row-vector can be stored in the registers too. But this approach is not generic and it is possible only with $n_g + n_g - l_{SIMD} < R_f$, with R_f the number of floating point registers.

Algorithm 5: Multi vectors/vector product with SIMD.

Data: n_g the number of result vectors to compute simultaneously (should be ≥ 2)
l_{SIMD} the number of floating point values proceed by the SIMD instructions
The size of the row-vector v or *SIZE_VEC* should be a multiple of the SIMD data type size l_{SIMD}
function `MultiVectorsVectorSimd(vec[`*SIZE_VEC*`], past[`$SIZE\_VEC + n_g - 1$`]) : res[`n_g`]`

```
register SIMD res_simd[n_g] = 0;
register value = load(vec[0]);
// We work first with values that are used once
for idxRes = 0 → l_SIMD-1 do
    res_simd[idxRes] = load(past[idxRes]) * value;
end
// We compute the rest of the res values and keep values in buffer
register buffer[n_g-l_SIMD];
for idxRes = l_SIMD → n_g-1 do
    buffer[idxRes-l_SIMD] = load(past[idxRes]);
    res_simd[idxRes] = buffer[idxRes-l_SIMD] * value;
end
past = @past[l_SIMD] ;
// For all values in the vec
for idxVec = l_SIMD → SIZE_VEC-1 by step l_SIMD do
    // Copy the current vec value
    value = load(vec[idxVec]);
    // Compute values that do not need shift
    for idxRes = 0 → Min(l_SIMD-1,n_g-l_SIMD) do
        res_simd[idxRes] = buffer[idxRes] * value;
    end
    // Compute values that can perform a shift
    for idxRes = l_SIMD → n_g-l_SIMD-1 do
        res_simd[idxRes] = buffer[idxRes] * value;
        buffer[idxRes - l_SIMD] = buffer[idxRes];
    end
    // Compute values that reload data
    for idxRes = n_g-l_SIMD → n_g-1 AND idxRes ≥ l_SIMD do
        buffer[idxRes-l_SIMD] = load(past[idxRes]);
        res_simd[idxRes] = buffer[idxRes-l_SIMD] * value;
    end
    // Compute values that do not reload data
    for idxRes = n_g-l_SIMD → n_g − 1 AND idxRes − l_SIMD < 0 do
        res_simd[idxRes] = load(past[idxRes]) * value;
    end
    past = @past[l_SIMD] ;
end
return SimdToScalar(res_simd) ;
```

3.3.3 Managing the Variation of the Row-Vectors

The slices are composed of contiguous NNZ values on their rows, but these row-vectors have different lengths.To achieve performance, the presented computational algorithms cannot have dynamic/runtime vector length v and solution size n_g. It is possible to compile different versions of the kernel for different n_g because it is fixed for an entire simulation. However, for the multi-vectors/vector kernels, we need to have an efficient way to choose among the different compiled versions when we process row after row in a slice. One solution is to compute several kernels and to have pointers of functions that we use for each row-vector depending on its length. One

of the drawbacks of this method is that the functions cannot be inlined and that it is a multi-steps access: load the row-vector length v, retrieve the corresponding function pointer and call it. Even if the hardware has some module to prepare such a call in advance, this is efficient only when the function pointer is always the same. That is why we end up with a different approach based on the switch-case statement. The idea is to compile different kernels and to call them from a *switch* that manages the different row-vector lengths. We put the function in the switch based on the hypothetical durations as shown in Algorithm 6: the cheapest function should be accessed very quickly whereas we agree to pay an extra-cost for the longest functions.

Algorithm 6: Efficient access to computational kernel.

Data: The current example uses SIMD data type of size 2 and supported by three kernels for row-vectors of lengths 2, 4 or larger.

```
function SliceComputation(values[], vec_lengths[NB_ROW], vec_starting[NB_ROW],
res[NB_ROW × n_g] ) :
    for idxRow = 0 → NB_ROW − 1 do
        switch vec_lengths[idxRow] do
            case 2 // Cheap access if length is 2
                res[idxRow, 0:n_g] += MultiVectorsVectorSimd_2( values, @past[vec_starting[idxRow]]) ;
                break;
            end
            case 4 // In case length is 4
                res[idxRow, 0:n_g] += MultiVectorsVectorSimd_4( values, @past[vec_starting[idxRow]]) ;
                break;
            end
            otherwise // Otherwise more expensive access for length multiple of 2
                res[idxRow, 0:n_g] += MultiVectorsVectorSimd_mod2( values, @past[vec_starting[idxRow]],
                vec_lengths[idxRow]) ;
            end
        endsw
        values = @values[vec_lengths[idxRow]] ;
    end
```

3.4 Implementation on GPU

3.4.1 Slice Computation on GPU

On CPU, it is possible to achieve performance by processing the row-vectors individually, but this kind of approach is no longer efficient on GPU because of its hardware particularities that we resume briefly before introducing two methods to compute a slice on this device. A more detail presentation of the GPU is provided in Appendix C.4. One can see a GPU as a many-threads device or a large SIMD processor unit. It executes several teams of threads, usually called *blocks*. In this paper, we explicitly refer to this set of threads as a thread-block to avoid the confusion with a block which is a sub-part of a matrix. The number of thread-blocks represents the dimension of the GPU grid. When running a GPU kernel, one has to choose the number of thread-blocks and the number of threads inside a thread-block. There are different levels of memory in most GPUs. The global memory can be accessed and shared by all the threads even from different thread-blocks, but it is slow and even slower if accesses are unaligned/uncoalesced inside a thread-block. The shared

memory is fast and shared among the threads of a thread-block, but it is very limited in size and can be slowed down because of bank conflicts. Finally, each thread has dedicated registers/local memory.

To provide a many-thread approach, we work with a block of values instead of vectors. We present two cut-out strategies that transform a slice in blocks, the Full-Blocking and the Contiguous-Blocking approaches. We call these pieces *blocks* and their dimension b_r (the number of rows) and b_c (the number of columns). In both cases b_c should be at least equal to d_{max} which is the longest row-vector in all the slices of the current simulation.

3.4.2 Full-Blocking Approach

In this approach, we extract and copy parts of the slices into blocks and leave the original structure of the values unchanged. A block contains the row-vectors that are inside a $2D$ interval of dimension b_r per b_c. If two row-vectors are separated by more than b_r rows or if they have some values separated by more than b_c columns, they cannot be in the same block. There are several algorithms to cut out a slice matrix into blocks. Our implementation is a greedy heuristic which has a linear complexity with respect to the number of rows in the slices. The algorithm starts by creating a block at the first row (for the first row-vector). Then, it progresses row by row and starts a new block whenever the current row-vector does not fit in the current block. Figure 3.12 shows an example of a cut-out using this algorithm and the resulting blocks. The generated blocks remain sparse in most cases. The algorithm needs a pair of integers for each block, which corresponds to the position of the upper-left values inside the original source slice as shown in Figure 3.12c. We have compared this algorithm to a dynamic block height b_r by increasing the blocks until a row-vector is out or even by using a dynamic programming approach to have as few blocks as possible (results are not included in this study). However, the extra-costs of seeking the best block configuration or having unpredictable block sizes are too significant.

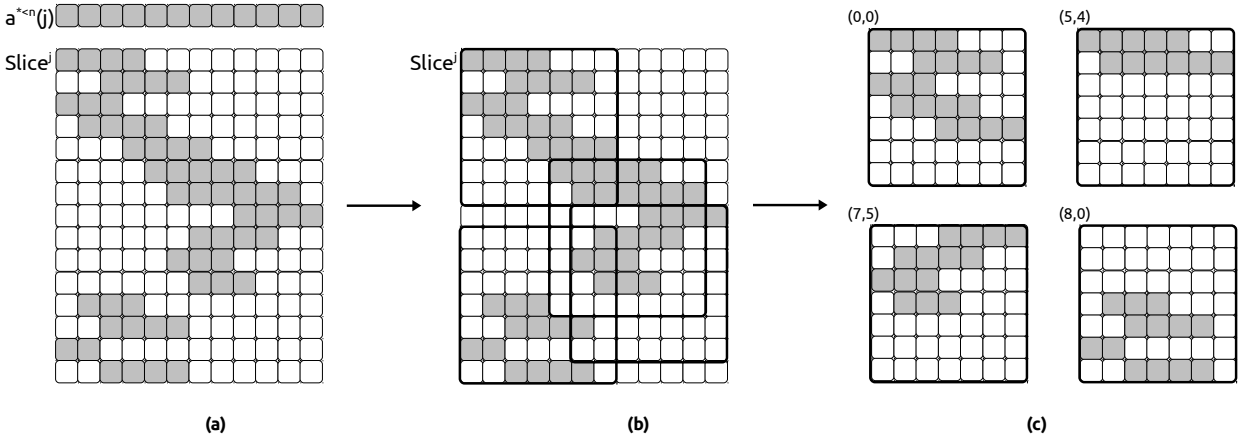


Figure 3.12: Example of a slice cut-out into blocks with the Full-blocking approach: (a) the original slice, (b) the blocks found by the greedy algorithm and (c) the blocks as they are going to be computed with their corner positions in the original slice. The block dimension is $b_c = 7 \times b_r = 7$.

The computation we have to perform is identical to the multi-vectors/vector product introduced in Section 3.2.3 and is called a multi-vectors/matrix product. It is also similar to a matrix/matrix

product with a leading dimension of one in the past values (which is not a matrix, but a special vector).

In this paragraph and in Figure 3.13, we give some implementation details of this operator on GPU. A thread-block of b_r threads is in charge of several blocks (all of dimension $b_r \times b_c$) from a slice interval. First, the threads copy the past values needed by a block in a shared memory array of size $b_c + n_g - 1$. Each thread treats one row per block and computes n_g results. The outer loop iterates b_c times over the columns of the block. The inner loop iterates n_g times to allow the threads to compute the result by loading a past value and using the block values. The n_g results are stored into local/register variables, which are written back to the main memory once a thread has computed its entire row. In this approach the threads read the block values from the global memory once and in a coalesced scheme column by column (the blocks are stored in column major). Also, n_g and b_c are known at compile time, thus the loops over the columns and the results can be unrolled. This is possible using *C++* templates; we compile lots of different kernels and select the appropriate one at runtime. There is no bank conflict because all the threads read the same values at the same time from the shared memory. Therefore, there are only broadcast operations.

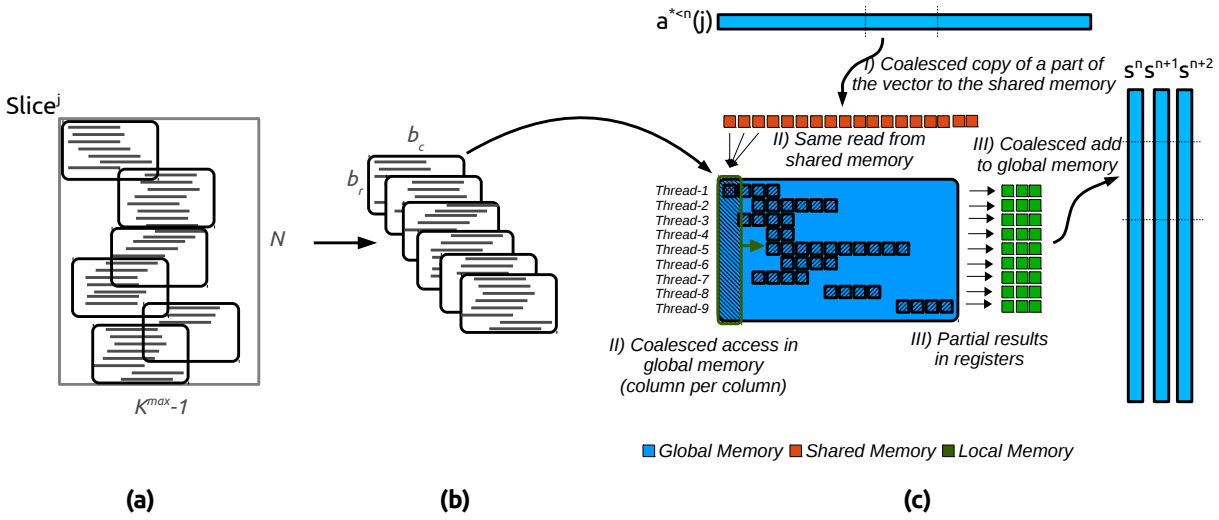


Figure 3.13: Example of the computation of a block from Full-blocking on GPU with $n_g = 3, b_c = 16$ and $b_r = 9$: (a) the original slice is split into blocks during the pre-computation stage, (b) the blocks are moved to the device memory for the summation stage and (c) a group of threads is in charge of several blocks and compute several summation vectors at the same time by performing a multi-vectors/matrix product. In $c-I)$ the threads copy the associate past values, in $c-II)$ each thread computes a row and in $c-III)$ threads add the results to the main memory.

The drawback of this method is the number of generated blocks and thus the extra-zeros padding. In the worst case, this method can generate one block of dimension $b_r \times b_c$ per row. Such configurations occur if b_r is equal to one or if each row-vector starts at a very different column compared to its neighbors. So b_c should be large enough to reduce the number of blocks, but the larger b_c is, the more zeros are used to pad. The numbering of the unknowns is also an important criterion, because the positions of the NNZ values, and thus the number of blocks, depend on it. In Section 3.7.4 we study the number of blocks generated for different numbering on a realistic test case. However, the main advantage of this method is that all rows in a block depend on the same past values.

3.4.3 Contiguous-Blocking Approach

In this approach, we do not keep the original structure of the values from a slice in the output blocks. We copy all the row-vectors into a block no matter where they start and where their values are positioned. In the Full-Blocking, all rows from a block have been copied from the same columns in the original slice. However, this is not guaranteed in the Contiguous-Blocking and each row of a block may come from different columns of the slices as shown in Figure 3.14. That is why we need to store the origin of the rows in the slices inside a vector to be able to compute them with the correct past values, see Figure 3.14c. It is possible to create several blocks per slice, but here we consider that $b_r = N$ and we create one block per slice.

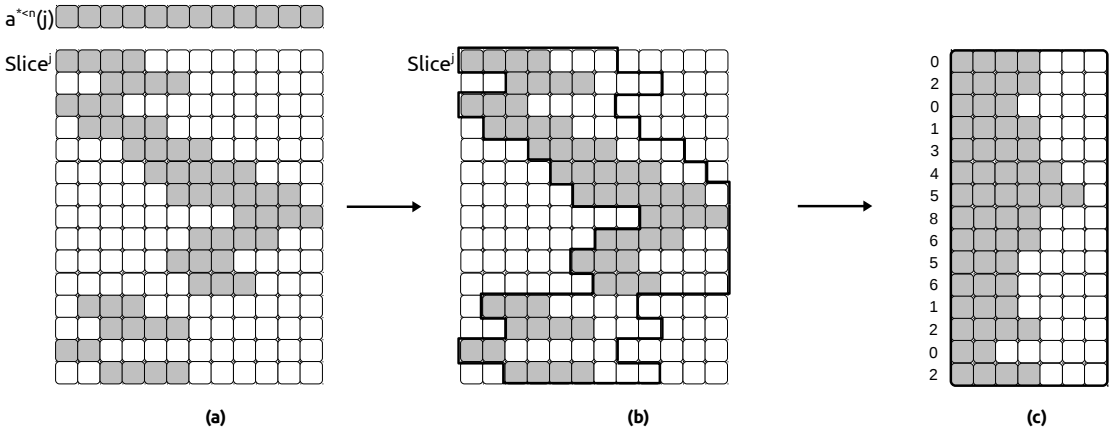


Figure 3.14: Example of a slice cutting-out into blocks with the Contiguous-Blocking approach: (a) the original slice, (b) the block build one row after the other and (c) the blocks as they are stored and computed with the starting point of each row in the original slice. The block dimension is $b_c = 7$ and $b_r = N$.

The kernel implementation of the Contiguous-Blocking is very similar to the Full-Blocking, except that it must take into account the column differences as shown in Figure 3.15. Instead of copying the past values needed for a block in the shared memory, the threads copy all the past values needed for a slice which is a vector of length $K^{max} + n_g - 1$. The threads of a thread-block do not access the same past values, but each thread accesses the past values that match the starting point of the row it has to compute. The threads continue to read the block values as they do in the Full-Blocking with a regular pattern. We may not be able to create a thread-block with N threads in it and in this case, some threads will be in charge of the computation of several rows of a block/slice. With this implementation, there might be some bank conflicts because the shared memory accesses are driven by the original column positions of the values: the threads might need the same values or different values from the same bank or different values in different banks.

The Contiguous-Blocking generates one block per slice. The number of columns in a block b_c must be at least equal to the longest row-vector d_{max} and there is no gain to have b_c greater than d_{max}. From the block size and the number of unknowns, we know the total number of values in the system generated by the Contiguous-Blocking: $N \times N \times b_c$. By calling d_{av} the average length of the row-vectors in the simulation, there are $N \times N \times d_{av}$ NNZ values, and the blocks are padded with $b_c - d_{av}$ zeros per row in average. The memory cost of this approach is $N \times b_c$ floating values and

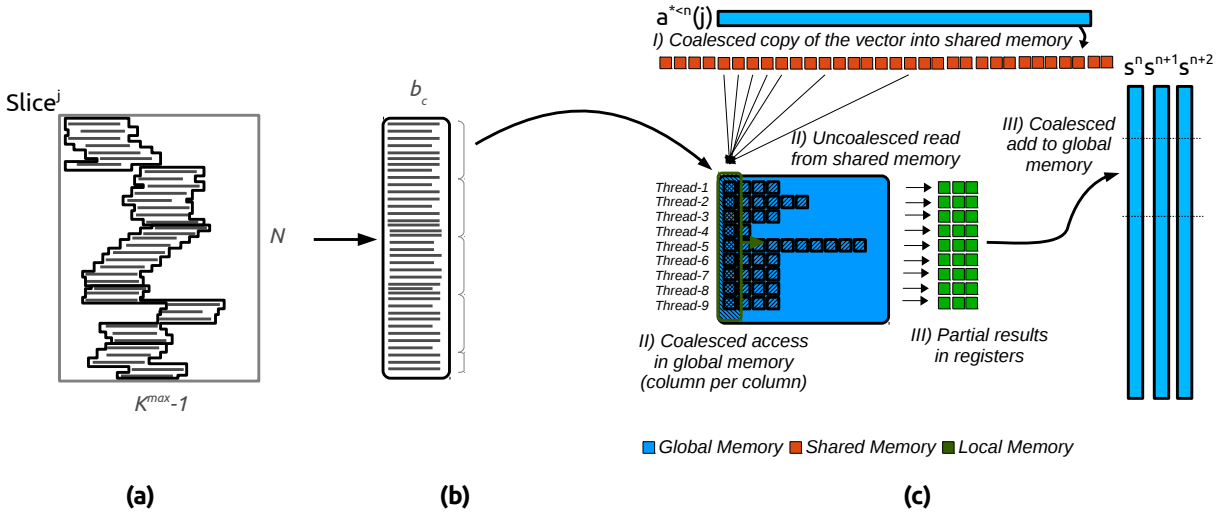


Figure 3.15: Example of the computation of a block from Contiguous-Blocking on GPU with $n_g = 3, b_c = 11$ and $nb - threads = 9$: (a) the original slice is transformed in a block during the pre-computation stage, (b) the blocks are moved to the device memory for the summation stage, (c) a thread-block is in charge of the blocks from a slice interval and computes several summation vectors at the same time by performing a multi-vectors/matrix product. In $c - I)$ the threads copy the past values associated to a slice, in $c - II)$ each thread computes a row and read the past values that match its own row and in $c - III)$ threads add the results to the main memory.

N integers per block generated from one slice, plus a copy of $K^{max} + n_g - 1$ floating values from the global to the shared memory and one matrix of $N \times n_g$ floating values for the results. The total number of Flop is $N \times b_r \times 2$, but the effective/real number of Flop is $N \times d_{av} \times 2$ per block.

3.5 Parallelization

3.5.1 Parallelization Strategy for Homogeneous Nodes

Shared memory parallelization. The straightforward parallelization in shared memory is implemented by splitting the slices' computation and the radiation between threads. This is done using OpenMP *for pragma* [6] but, to better balance the workload over the threads, we assign to each of them the same amount of data. Finally, most linear solvers do not support shared memory parallelization and in such a case the call to the solve routine is done by a single thread. No optimizations are done to manage the NUMA effects that happen when a processor accesses to its non-local memory, see Appendix C.1 for more details.

Distributed memory parallelization. The parallelization over distributed memory is done using Message Passing Interface (MPI) [7] and we name a MPI process a *process*. In our implementation, each process is responsible of several slices inside an interval. This interval from A_i to B_i can be obtained in different ways: for example, by dividing the number of N slices equally or by taking into account the amount of work in each slice. Each process needs to have the past values of the unknowns which match its slice interval. In a first stage, each process computes a part of the summation vectors without communicating with others. Then, all processes synchronize and call a linear solver to solve Equation 1.5 and obtain the current solution a^n. This algorithm is presented in a schematic view in Figure 3.16.

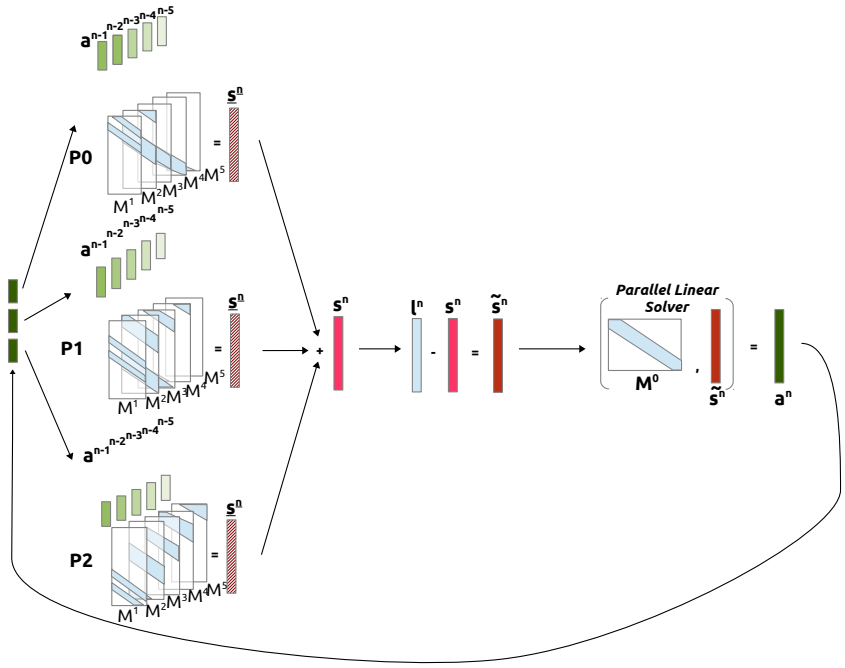


Figure 3.16: Parallel Solver Schematic View

However, from our implementation of the multi-vectors/vector product, we compute several time steps together followed by a radiation stage. Therefore, there is an inner loop to do the radiation while the outer loop progresses by n_g steps as shown in Figure 3.17.

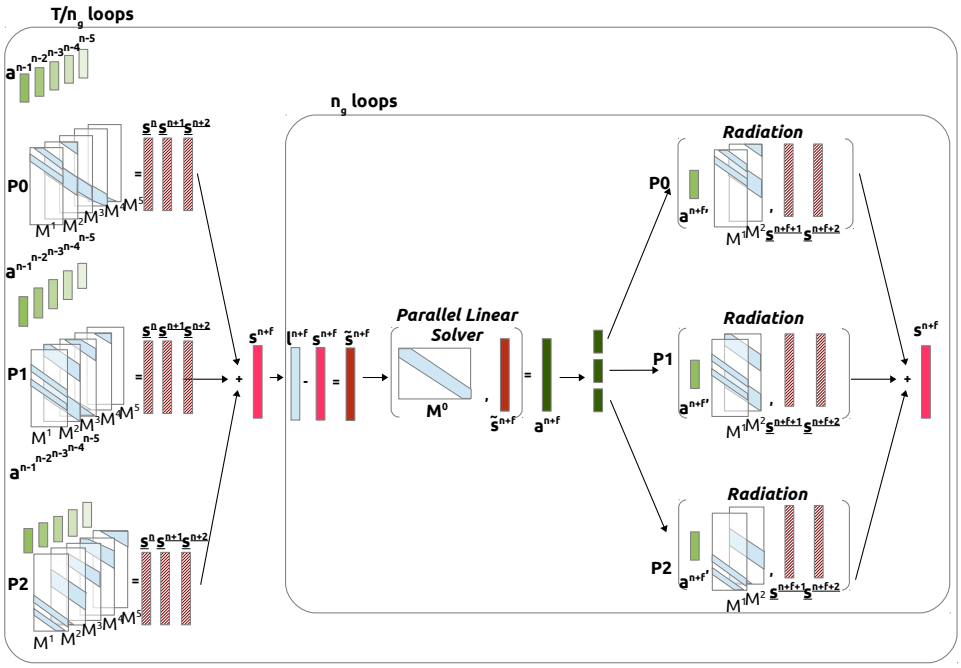


Figure 3.17: Parallel Solver with Multiple Time Steps Schematic View

The complete algorithm is written in Algorithm 7. If the number of threads per process is 1 and the parallelism relies on MPI only, we refer this implementation as the Full-MPI. If the number of threads is greater than 1, we refer to it as the Hybrid-MPI/OpenMP implementation. From the algorithm, we remark that, at each iteration, the current result a^n is saved to disk for later work,

and it also has to be distributed on each process to give them the current result for their interval j_{start} to j_{end}.

Algorithm 7: Complete simulation with Hybrid-MPI/OpenMP parallelization

Data: $Slices[N]$ the interaction matrices in slice/vectors shape. Each process works on an interval [j_start; j_end] that cover the entire slices.
Result: $PastValues[j\_end - j\_start + 1][NB\_STEPS + n_g - 1]$ the state of the unknowns for all time step
begin
  // Linar Solver initialization (factorize/inverse M^0)
  $invM^0$_handle = linear_solver($M[0]$);
  // For all time step with progression by n_g
  for $n = 0 \rightarrow NB\_STEPS\text{-}1$ *by* n_g **do**
    S[n_g][N] = 0;
    // Compute n_g vectors with each slices in my interval
    #pragma omp parallel reduce(+:S);
    for j = j_start $\rightarrow$ j_end **do**
      foreach Vec v in Slices[j].blocks **do**
        S[:][v.row] += MultiVectorsVector(v.values, PastValues[j][v.col - n_g + 1 :v.col + v.length]) ;
      end
    end
    // Finalization
    for idx = 0 $\rightarrow$ n_g-1 **do**
      distributed_reduce(S[n_g - idx -1][:]);
      a^n = solve($invM^0$_handle, L[n+idx][:] - S[n_g - idx - 1][:]);
      master saves a^n to disk;
      // Copy result in Pastvalues format
      PastValues[j_start:j_end][NB_STEPS - n - 1] = a^n[j_start:j_end];
      // Radiation
      #pragma omp parallel;
      for past = idx + 1 $\rightarrow$ n_g-1 **do**
        S[n_g - past][:] += SpMV($M^{past-idx}$[j_start:j_end],a^n[j_start:j_end];
      end
    end
  end
end

3.5.2 Parallelization Strategy for Heterogeneous Nodes

A *worker* defines a processing unit or a group of processing units on the same node. We dedicate one CPU core to manage one GPU and to be in charge of the kernel calls and the data transfers between the host and the device. Thus, a GPU-worker is a couple of CPU/GPU. All the cores from a node that are not in charge of a GPU are seen as a single entity called the CPU-worker. So one node is composed of one GPU-worker per GPU and one single CPU-worker as shown in Figure 3.18. All the workers take part in the summation step, but only the CPUs are involved during the factorization of M^0 and the solves. Inside a CPU-worker, we balance the work between threads as we do with nodes by assigning the same amount of data to each thread.

Dynamic Balancing between Heterogeneous Workers

A node i is responsible of the slices in the interval $[A_i; B_i]$ and computes a part of the summation s^n, see Equation (1.3). However, this interval has to be balanced between the different workers

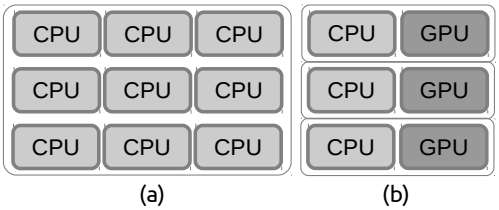


Figure 3.18: Example of workers in a node composed of 12 CPU and 3 GPUs: (a) the CPU-worker composed of all the CPUs that are not in charge of a GPU, (b) three GPU-workers each composed of a couple of CPU/GPU.

within the node. We constrain each worker to have a contiguous interval of data/slices which enables copying and moving in a single call. Furthermore, workers can have distinct computational capacities, and the slices can have distinct costs. Therefore, the problem is to find the optimal interval for each worker that covers the node slices and with the minimum wall time. The wall time is the maximum time taken by a worker to compute its interval.

One possibility to solve this problem is to perform a calibration and to estimate the speed of each worker. However, such an approach takes a non-negligible time, and it is difficult to consider all the possible configurations and data transfers. We could also perform a warm-up stage and have each worker computes the full slice interval to know the computation time taken for each worker for each slice. Not only, this process can be extremely slow, but the fastest worker for each slice individually may not be the fastest to compute a given interval. In fact, GPUs are much more limited by their memory than CPUs, and if we assign a slice interval to a GPU that does not fit in its memory, it will induce very slow copies (from CPU to GPU or even from hard-drive to GPU). We propose an heuristic to balance the work after each iteration in order to improve the next ones. There are plenty of methods that can perform such an operation, and we propose a greedy algorithm with a $\Theta(W)$ complexity, where W is the number of workers.

The algorithm we propose considers that the average time t_{avg} of all workers in the previous iteration is the ideal time and the objective for the next iteration. For the first iteration, we assign the same number of slices to each worker. At the end of the iteration, each worker w_i has computed its complete interval of slices $[a_i; b_i]$ in time t_i. We do not measure the time taken for each individual slice, but we have an estimation of the cost $c_i = t_i/s_i$, with $s_i = b_i - a_i + 1$ as the number of elements computed by the worker of index i.

Workers that were slower than average (if $t_{avg} < t_i$) should reduce their intervals. However, the faster workers (if $t_i < t_{avg}$) should increase their intervals and compute more slices. We consider that each slice on a given worker has the same cost. Therefore, for a slow worker w_i we remove $r_i = (t_i - t_{avg})/c_i$ slices from its interval. We would like to do the same for a faster worker and add $o_i = (t_{avg} - t_i)/c_i$ to its intervals. But in most cases, the number of elements to remove from the slower workers is not equal to the number of elements we want to give to the faster workers. For example, in a system with two workers and the following properties in the previous iteration: worker w_1 has computed $s_1 = 10$ elements in $t_1 = 10s$ and worker w_2 has computed $s_2 = 3$ elements in $t_2 = 4s$. The average execution time is $t_{avg} = (10 + 6)/2 = 8s$ and the first worker should remove $r_1 = (10 - 8)/(10/10) = 2/1 = 2$ elements, whereas the second worker should

increase its interval by $o_2 = (8-4)/(4/3) = 4/(4/3) = 3$ elements.

Thus, the faster workers have to share the slices that have been removed. We sum the number of available slices $S_{removed} = \sum r_i$ and the number of required slices $S_{given} = \sum o_i$ and distribute them using a weight coefficient. A faster worker will have its interval increased by $o_i \times S_{removed}/S_{given}$ which guarantees an equality between the slices removed and given. The number of slices to compute is updated for each worker ($s_i = s_i - r_i$ or $s_i = s_i + o_i \times S_{removed}/S_{given}$) and then a new contiguous slice interval is assigned to each worker. An example of this heuristic is presented in Figure 3.19.

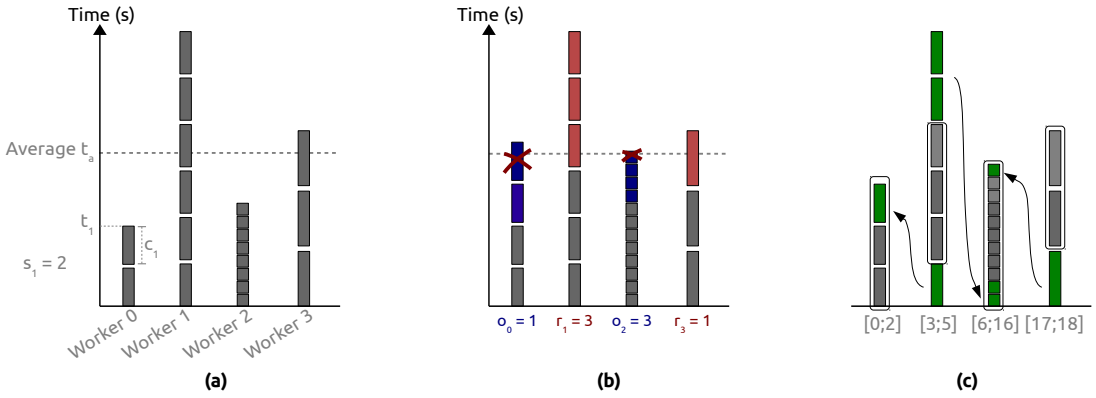


Figure 3.19: Illustration of one iteration of the balancing algorithm with 4 workers. Workers 1 and 3 are above the average and should drop 3 blocks and 1 block, respectively (red blocks). Workers 0 and 2 are under the average and should acquire $1/4$ and $3/4$, respectively of the dropped blocks (blue blocks).

This algorithm should find an optimal solution in one iteration if there is the same amount of work per element. However, there is no guarantee that the algorithm can find the optimal solution or even that it improves as it iterates. Theoretically, we can stop the algorithm if a configuration does not give a better result than the previous one, but in practice, some latency or unpredictable events make this statement unsafe. Therefore, in practice we stop the algorithm after a given number of iterations and rollback to the best configuration that was generated.

3.5.3 Parallel Linear Solvers Considerations

The costly operation is the summation stage, but the solve of M^0 should not be neglected. In fact, the number of unknowns may be small, and state-of-the-art solvers are made to manage matrices with dimension of several millions of unknowns. Moreover, the solve and the obtaining of the result a^n are the only operations that use communication between nodes, which makes them critical to scale among a large number of nodes.

Since the matrix M^0 is sparse it is natural to rely on a sparse solver. We use it as a black-box and it is in charge of managing the matrix ordering, the communications and the data distribution.

While it can seem unnatural to use a dense solver in our case, the question should be asked. In fact, in our problem, we perform a lot of solves using the same matrix M^0 and, as stated previously, the dimension of this matrix is small for sparse solvers. From this key-point, we may ask whether a dense solver will finally scale more than a sparse solver and at a which point it will become competitive. In addition, if we compute the inverse $M^{0^{-1}}$ we reduce drastically the number

of communication at each turn, which is critical as the number of nodes increases. In fact, the processes do not need the entire a^n result but only the part that match their slice interval.

After the summation stage, each node hosts a partial result s^n. The processes can divide the application of illumination l^n and then perform a reduction all-to-all to have all processes holding $\widetilde{s}^n$. Afterwards, each process multiplies this vector by its part of $M^{0^{-1}}$ and obtains a sub-vector of a^n that matches its slice interval. The processes are then able to perform the radiation or the future slice computations. Therefore, it makes the processes communicate only once per iteration. But we must inverse the matrix once at the beginning of the application. Moreover, the M^0 matrices are usually well conditioned which leads to a stable accuracy in most algorithms.

3.5.4 Division of the summation (Far Field Near Field)

The call to the linear solver is the only communication point in our strategy and that is why we try to develop an algorithm that hides this potential bottleneck. One possibility is to perform a call to the linear solver on one side and to compute a part of the summation on the other side; we can only compute a partial summation vector s^{n+1} since the current result will be obtained after the solve. Once these two tasks are finished, we have a^n and a partial s^{n+1}. We can compute the radiation stage to obtain the complete s^{n+1} and we process it with the linear solver. Therefore, it is one way to hide the communication, and it can be done using two main strategies.

If we think of having two threads T^1 and T^2 in charge of the simulation, the first strategy can be expressed as follows. T^1 calls the linear solver using s^n to obtain a^n in time t_{Solve} while T^2 is performing the summation to compute a partial s^{n+1} in time $t_{Summation}$. Then both threads are involved in the radiation to generate s^{n+1} in $t_{Radiation-Par}$. After this point, we are back to the initial state, and the threads are divided again and the time for one iteration is given by $Max(t_{Solve}, t_{Summation}) + t_{Radiation-Par}$.

In the second strategy T^1 is in charge of the radiation alone. From the previous iteration, s^n is a partial summation, T^1 performs the radiation in $t_{Radiation-Seq}$ and then calls the linear solver using s^n to obtain a^n while T^2 is performing the summation to compute a partial s^{n+1}. In this case, there is a simple join between both threads, and the execution time is $Max(t_{Solve} + t_{Radiation-Seq}, t_{Summation})$.

However, the algorithm becomes more complicated in our case because we compute several summations together, see Section 3.2.3. In this case, on one hand we compute a^n to a^{n+n_g-1} and on the other hand the partial s^{n+n_g} to s^{n+2n_g-1}. This makes the radiation expensive and thus the first strategy is certainly to be privileged because we expect to have $Max(t_{Solve}, t_{Summation}) + t_{Radiation-Par} < Max(t_{Solve} + t_{Radiation-Seq}, t_{Summation})$. In practice the result are better than the usual approach but this might be an interesting approach in case of extremely high number of nodes.

3.6 Building the Interaction Matrices

As stated in the context, our solver is a layer of an application and our study does not cover the generation of the interaction matrices. But our solver must call the module responsible for this work during an initialization stage. We can estimate the slice interval from the problem dimension, and since the matrices are symmetric positive definite, we can reduce the generation time by dividing

the work among the threads/processes. Figure 3.20 shows how this generation is divided among the nodes and how these ones need to communicate to make all the nodes having their slices. However, a balancing stage is necessary before beginning the simulation to have around the same amount of data per nodes.

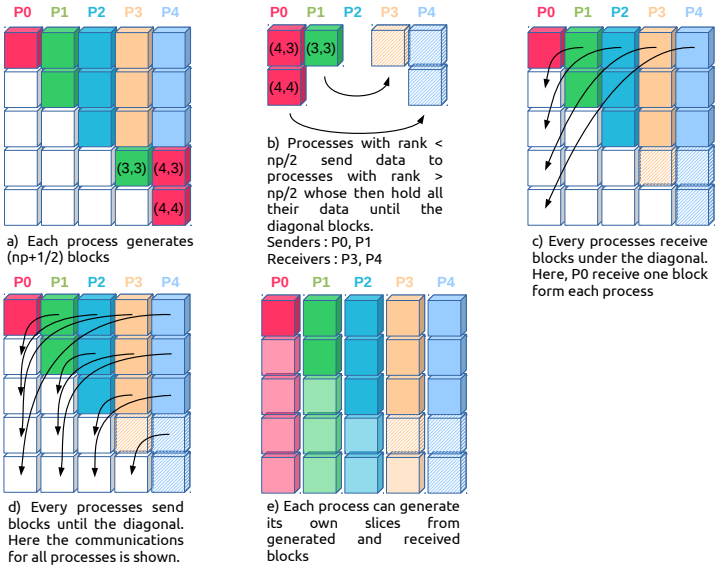


Figure 3.20: Interaction Matrices Generation on distributed memory.

3.7 Performance and Numerical Study

3.7.1 Experimental Setup

We use different configurations in terms of hardware (CPU/GPU) and libraries (Gcc/MPI) on the Plafrim test-bed. In all cases, parallelization over nodes is supported by MPI [7] and parallelization inside nodes over shared memory by OpenMP [6]. The calculations are performed in Single (32-bit arithmetic) or Double (64-bit arithmetic).

SSE Configuration

We use up to 32 nodes and each node has the following configuration: 2 Quad-core Nehalem Intel® Xeon® X5550 at $2.66GHz$ and $24GB$ (DDR3) of shared memory. Peak performances are $21.28\,GFlop/s$ in single and $10.64\,GFlop/s$ in double for one CPU core. We use the Gcc 4.7.2 compiler and Open-MPI 1.6.5. The compilation flags are -m64 -march=native -O3 -msse3 -mfpmath=sse. The direct solver is a state of the art solver Mumps 4.10.0 [98] which relies on Scotch 5.1.12b.

SSE/Tesla Configuration

We use up to 8 nodes and each node has the following configuration: 2 Hexa-core Westmere Intel® Xeon® X5650 at $2.67GHz$ and $36GB$ (DDR3) of shared memory and 3 NVIDIA Tesla M2070 GPU ($1.15GHz$), 448 Cores, $6GB$ (GDDR5) of dedicated memory. Peak performances are

21.36 $GFlop/s$ in single and 10.68 $GFlop/s$ in double for one CPU core and 1.03 $TFlop/s$ in single and 515 $GFlop/s$ in double for one GPU. The nodes are connected by an Infiniband QDR 40 Gb/s network. We use the GCC 4.7.2 compiler, OpenMPI 1.6.5 library and CUDA SDK 5.5. The compilation flags for GCC are -m64 -march=native -O3 -msse3 -mfpmath=sse and for NVCC -arch=sm_20 -use_fast_math. The direct solver is also Mumps 4.10.0 which relies on Scotch 5.1.12b.

AVX Configuration

We use up to 20 nodes and each node has the following configuration: 2 Dodeca-core Haswell Intel® Xeon® E5-2680 at $2,50GHz$ and $128GB$ (DDR4) of shared memory. Peak performances are 40 $GFlop/s$ in single and 20 $GFlop/s$ in double for one CPU core and the nodes are connected by an Infiniband QDR 40 Gb/s network. We use the GCC 4.9.2 compiler and OpenMPI 1.8.4 library . The compilation flags for GCC are -m64 -march=native -O3 -mavx -mfmad . The direct solver is Mumps 4.10.0, and relies on Scotch 6.0.4.

AVX/Kepler Configuration

We use up to 5 nodes and each node has the following configuration: 2 Dodeca-core Haswell Intel® Xeon® E5-2680 at $2,50GHz$ and $128GB$ (DDR4) of shared memory, as in the $AVX-Configuration$, and 4 NVIDIA Kepler K40M GPU ($745MHz$), 2880 Cores, $12GB$ of dedicated memory. Peak performances are 40 $GFlop/s$ in single and 20 $GFlop/s$ in double for one CPU core and 4.29 $TFlop/s$ in single and 1.43 $TFlop/s$ in double for one GPU. The nodes are connected by an Infiniband QDR 40 Gb/s network. We use the GCC 4.9.2 compiler, OpenMPI 1.8.4 library and CUDA SDK 7.0.28. The compilation flags for GCC are -m64 -march=native -O3 -mavx -mfmad and for NVCC -arch=sm_30 -use_fast_math. The direct solver is Mumps 4.10.0, and relies on Scotch 6.0.4.

3.7.2 Balancing Quality Study

In Section 3.5.2, we describe the heuristic we use to balance the summation work among the workers inside a node. In Table 3.3, we test this balancing method by emulating different cases to study how far from the optimal distribution we are. We do not perform real simulations, but we rather generate several configurations composed of two arrays. The first array represents the workloads, which is the cost per element, and we refer to it as the *Work Distribution* (an element can be seen as a slice in our study). The second array contains the performance of the workers, which is the time taken per worker to compute a unit of work, and we refer to it as *Worker Heterogeneity*. To get a worker virtual execution time we multiply the sum of the costs from the interval that is assigned to this worker per its performance factor. The virtual wall time of a distribution is obtained by taking the maximum virtual execution time from all workers involved. The balancing algorithm is executed during a given number of iterations and compared to the optimal choice. We find the optimal balance using dynamic programming, that is: finding the contiguous interval for each worker that has the minimum wall time. The results show that the balancing algorithm is

close to the optimal even when the cost per element or the worker performance factors are very heterogeneous.

Work distribution\Worker heterogeneity				
Up-Down 	0.0%	0.1%	0.1%	0.0%
Up-Up 	0.0%	0.0%	0.1%	0.1%
Up 	0.1%	0.1%	0.2%	0.2%
Random 	0.0%	0.0%	0.2%	0.0%
Stable 	0.0%	0.0%	0.1%	0.1%

Table 3.3: Balancing Algorithm Vs. Optimal Choice. Extra-cost of the dynamic balancing algorithm against the optimal choice after 40 iterations with 6 workers and $10,000$ elements. A zero score means that the optimal choice has been achieved.

3.7.3 Sequential Flop-rate

CPU Multi-vectors/vector Product

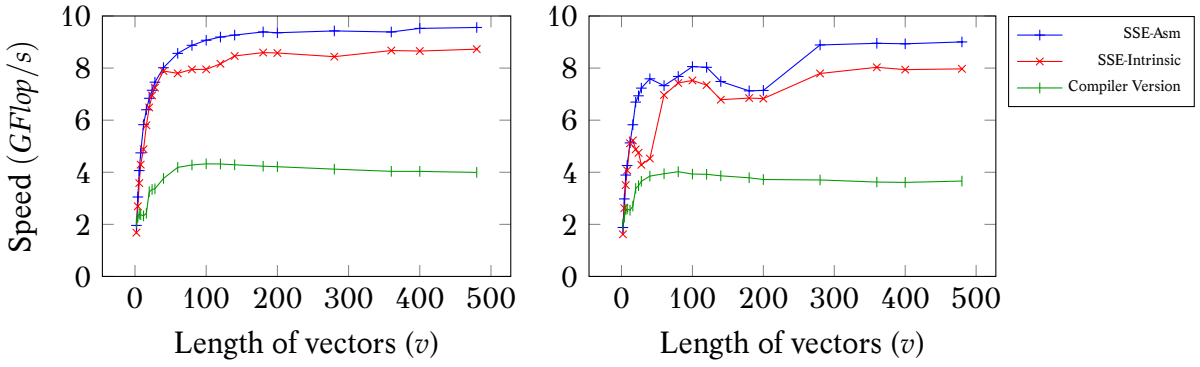
We compare several implementations of the multi-vectors/vector product for the $SSE-Configuration$ and $AVX-Configuration$. We have $n_g = 8$ as it is enough to bypass the memory bandwidth limitation without paying too much extra cost in the radiation stage and there are enough floating registers to apply the assembly optimizations.

The first implementation comes out of the Equation (3.2) and is implemented in C. Some important compilation flags are used to enable loop unrolling and the use of SSE or AVX instructions by the compiler, but no manual use of SIMD is made. This is referred to as the Compiler Version implementation.

The second version is written in C and comes out of Section 3.3.1. It is written with intrinsic SSE functions proposed by the compiler and SSE data types ($\_\_m128(d)$). We refer to it as the SSE-Intrinsic implementation. A similar implementation is made using AVX intrinsics ($\_\_m256(d)$) and is called AVX-Intrinsic.

We have analyzed the assembly code the compiler has generated, and we have considered that it is not optimal for both implementations. Thus, we have developed a third implementation in asm64 assembly to maximize the data re-use as explained in Section 3.3.2. With $n_g = 8$ it is possible to use all 16 SSE registers to read each value only once from the main memory. We refer to it as the ASM-SSE and ASM-AVX implementations. We developed an AVX-Template kernel which use AVX intrinsics and with a template n_g; this kernel can be compiled for any n_g length and relies on the Algorithm 5.

Figure 3.21 shows the Flop-rate in Double precision for different lengths of vector v using the $SSE-Configuration$. The two SSE based implementations are close, but the ASM-SSE can achieve a slightly higher Flop-rate for large vectors. Both implementations suffer from small cache effects for $N_r = 1\,000$ and $v = 100$ (Figure 3.21a) and for $N_r = 20\,000$, $v = 25$ and $v = 80$ (Figure 3.21b).

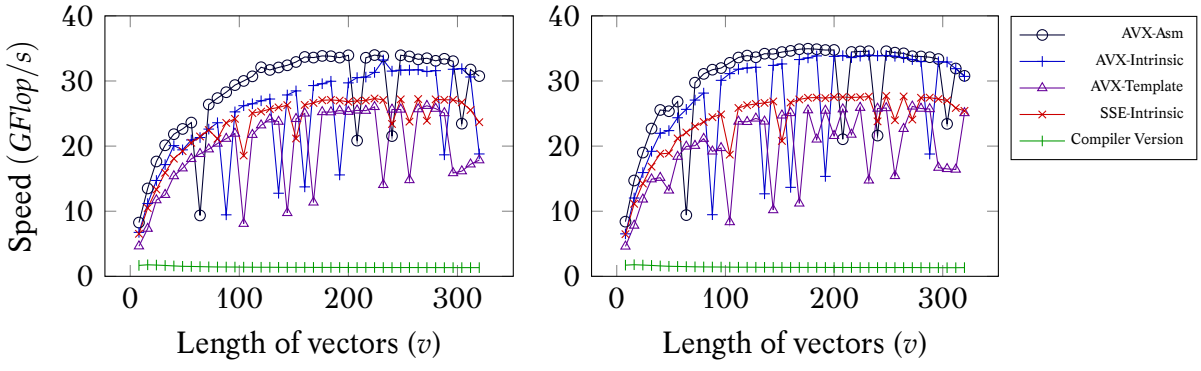


(a) Number of rows in slices $N_r = 1\,000$

(b) Number of rows in slices $N_r = 20\,000$

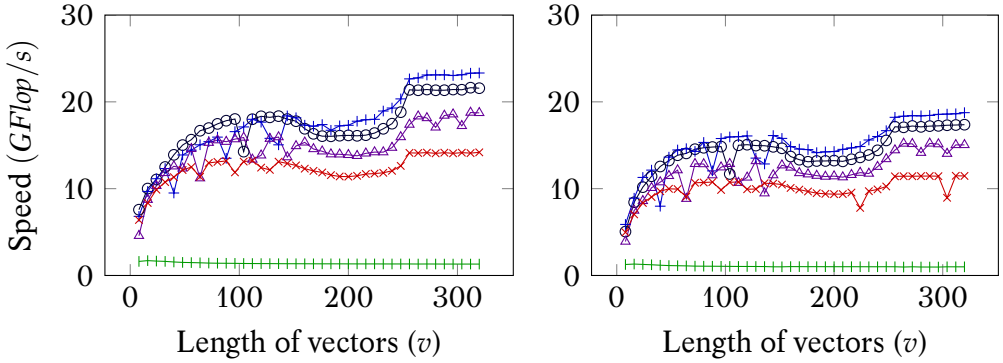
Figure 3.21: Performance evaluation in $GFlop/s$ for the multi-vectors/vector slice computation code in Double precision for three implementations with $n_g = 8$ and the $SSE - Configuration$. The test cases are slices of dimension $N_r \times v$.

Figure 3.22 shows the Flop-rate for different lengths of vector v using the $AVX - Configuration$ in Single and Double precision. This time the ASM-AVX version performs better in Single but the AVX-Intrinsic version is better in Double. The AVX-Template is efficient but in Single it is less efficient compared to SSE-Intrinsic. However, among these implementations, only the C-Compiler and AVX-Template can be compiled for any n_g. We see that all the SIMD version are suffering of cache effects: in Single precision the speed decreases for $v = 200$, Figure 3.22a and Figure 3.22b, and in Double precision for $v = 100$, Figure 3.22c and Figure 3.22d.



(a) Number of rows in slices $N_r = 1\,024$, in Float

(b) Number of rows in slices $N_r = 20\,480$, in Float



(c) Number of rows in slices $N_r = 1\,024$, in Double

(d) Number of rows in slices $N_r = 20\,480$, in Double

Figure 3.22: Performance evaluation in $GFlop/s$ for the multi-vectors/vector slice computation code for five implementations with $n_g = 8$ and the $AVX - Configuration$. The test cases are slices of dimension $N_r \times v$.

The length of the vectors of the slices in real test cases depends on Δt the time step, and the size of the elements on the mesh. In the airplane test case, each vector has a length between 1 and 15 and the average length is 9.5. In this configuration, the ASM-SSE implementation achieves 3.9 $GFlop/s$ per core (Compiler Version achieves 1.7 $GFlop/s$) for a peak performance of 10.64 $GFlop/s$.

GPU Full-Blocking Kernel Flop-rate

Table 3.4 presents the Flop-rate of the GPU and the CPU implementations for the blocks generated by the Full-Blocking method described in Section 3.4.2 using the $SSE/Tesla - Configuration$. Table 3.5 shows the same study but using the $AVX/Kepler - Configuration$. We look at the Flop-rate achieved by one GPU or one CPU-core for different sizes of dense blocks. We create thread-blocks of b_r threads and the best dimension of the grid (the number of thread-blocks) depends on the size of the block and on the number of blocks. The performance increases with the block size on GPU because increasing b_r increases the number of threads in a thread-block and increasing b_c provides more work to each thread and allows to unroll larger loops. The CPU implementation benefits also from the instruction pipelining when we increase b_r or b_c, but its performance decrease when the block exceeds a size ($b_c > 16$ and $b_r > 32$) due to cache effect. We remind that there are zero values in the blocks and that increasing the size of the blocks should reduce the number of generated blocks, but it also increases the zero padding. So finding the correct dimension is a matter of finding the fastest kernel for the generated number of blocks (% of NNZ).

		Width (b_c)							
		16	32	64	128	16	32	64	128
	Height (b_r)	GPU				CPU			
Single	32	53	64	105	139	2.9	8.4	8.8	8.2
	64	124	176	230	270	8.8	7.5	6.7	6.3
	128	138	196	244	280 (27%)	8.2	7	6.4	6 (28%)
Double	32	39	51	67	68	2.5	3.6	3.4	3.5
	64	68	95	128	136	3.4	3.1	3	2.8
	128	73	100	128	141 (27%)	3.8	3.1	3	2.9 (27%)

Table 3.4: Performance of computing the blocks from Full-Blocking. Performance in $GFlop/s$ for 420 slices composed of 6400 rows and b_c columns and $n_g = 8$. % percentage of the Peak performance against single|double: GPU 1.03 $TFlop/s$|515 $GFlop/s$, CPU 21.36 $GFlop/s$|10.68 $GFlop/s$ ($SSE/Tesla - Configuration$).

GPU Contiguous-Blocking Kernel Flop-rate

Table 3.6 presents the Flop-rate of the GPU and CPU implementations for the blocks generated by the Contiguous-Blocking method described in Section 3.4.3 using the $SSE/Tesla - Configuration$. Table 3.6 shows the same study but for the $AVX/Kepler - Configuration$. We have no choice in the block size parameters: b_r is set to N and b_c to d_{max}. This is a drawback for the CPU version because it leads to a large block and a high leading dimension between columns but we agree to pay this extra cost since our goal is to concentrate on the GPU. This method has better performance for the GPU against Full-Blocking which means that it does not pay any extra-cost for its irregular/uncoalesced

		Width (b_c)							
		16	32	64	128	16	32	64	128
	Height (b_r)	GPU				CPU			
Single	32	58	86	103	121	4.8	4.8	4.8	4.7
	64	105	161	194	228	7.5	7.5	7.7	7.6
	128	140	219	303	372 (8%)	8.4	8.4	8.3	8.4 (21%)
Double	32	46	66	77	93	4.3	4.3	4.4	4.3
	64	79	116	138	167	4.6	4.7	4.7	4.6
	128	89	137	187	231 (16%)	5.6	5.7	5.7	5.7 (28%)

Table 3.5: Performance of computing the blocks from Full-Blocking. Performance in $GFlop/s$ for 420 slices composed of 6400 rows and b_c columns and $n_g = 8$. % percentage of the Peak performance against single|double: GPU $4.29\ TFlop/s|1.43\ GFlop/s$, CPU $40\ GFlop/s|20\ GFlop/s$ ($AVX/Kepler - Configuration$).

shared memory accesses. Moreover, this implementation copies all the past values needed by an entire slice in the shared memory which is an advantage compared to the GPU Full-Blocking because it copies past values for each block (smaller copies but more frequent).

	Width (b_c)							
	16	32	64	128	16	32	64	128
	GPU				CPU			
Single	146	203	251	285 (28%)	2.7	3	2.3	2.3 (11%)
Double	82	113	139	158 (31%)	2	1.6	1.6	1.4 (13%)

Table 3.6: Performance of computing the blocks from Contiguous-Blocking. Performance in $GFlop/s$ for 420 slices composed of 6400 rows and b_c columns and with 14 GPU thread-blocks of 512 threads and $n_g = 8$. % percentage of the Peak performance against single|double: GPU $1.03\ TFlop/s|515\ GFlop/s$, CPU $21.36\ GFlop/s|10.68\ GFlop/s$. The CPU version is an C optimized by the compiler ($SSE/Tesla - Configuration$).

	Width (b_c)							
	16	32	64	128	16	32	64	128
	GPU				CPU			
Single	243	338	431	496 (11%)	4.3	5.5	7.8	6.8 (17%)
Double	143	199	248	286 (20%)	3.9	5.6	4.2	4.3 (21%)

Table 3.7: Performance of computing the blocks from Contiguous-Blocking. Performance in $GFlop/s$ for 420 slices composed of 6400 rows and b_c columns and with 15 GPU thread-blocks of 1024 threads and $n_g = 8$. % percentage of the Peak performance against single|double: GPU $4.29\ TFlop/s|1.43\ GFlop/s$, CPU $40\ GFlop/s|20\ GFlop/s$ ($AVX/Kepler - Configuration$).

Warp occupancy and performance of the Contiguous-Blocking kernel. In Section 3.4.1 we give an overview of the GPU specifications, but to achieve performance with this architecture we also need to focus on hardware details. The current discussion is based on the $SSE/Tesla - Configuration$ GPU but the same principle applies to the $AVX/Kepler - Configuration$ GPU. With the $M2070$, a thread-block can use a maximum of 32768 registers and 49152 bytes of shared memory and supports 1536 threads. This GPU has 14 multiprocessors each with a warp of size $wp = 32$ and one can see this GPU as 448 CUDA Cores. However, there is no concurrency inside a warp. Using this information, we have to fill all the multiprocessors by choosing the number of thread-blocks

and the number of threads. The CUDA compiler (NVCC) gives us the memory occupancy of our kernel. Each thread uses 62 registers, and one thread-blocks needs 6104 bytes of shared memory to store the past vector of length $K^{max} + n_g - 1$ (with $K^{max} = 756$ and $n_g = 8$). The size of the memory occupancy in the shared-memory is not related to the number of threads and the available shared-memory does not appear as a limitation. On the other hand, the number of available registers limits the number of threads to $max_{th} = 32768\, div\, 62 = 528$ per thread-block. If we use more threads than max_{th}, the registers become insufficient and the kernel will use the local memory as a replacement which is drastically slower. It is recommended that the number of threads nb_{th} in a thread-block is a multiple of the warp size wp; thus, for our configuration $nb_{th} = 512 \leq 528$ threads. Choosing nb_{th} not to be a multiple of wp means that some multiprocessors will be under-exploited (some CUDA cores will remain idle). We have implemented our kernel to have low registers and shared memory usage but also low divergence between threads. In fact, divergence between threads is a costly operation since it requires to perform all the different code paths of the threads inside a warp separately. We have two divergences that may happen during the copy of the past vector into the shared memory or during the loop over the rows. When copying the past vector, the threads with $id < (K^{max} + n_g - 1) \bmod nb_t$ perform one extra copy compared to the others. Since the number of threads may not be a multiple of the number of row N, some threads could perform more computation. Finally, all accesses to the global memory are coalesced or similar inside a thread-block. The GPU hides the global memory latency by swapping threads on the multiprocessors, while a warp of threads is requesting a memory access. Since we have 512 threads per thread-block, we just need to have 14 thread-blocks to ensure that the 14 GPU multiprocessors are filled. Having more than 14 thread-blocks is not needed as it could deliver worse performance because of the costs for the GPU to manage thread-blocks and increase the reduction work of the result matrices. The source code of the Contiguous-Blocking kernel implementation is given in Appendix C.4.

3.7.4 Test Case

We now consider a real simulation to study the parallel behavior of the application. Our test case is the airplane presented in Section 1.2.4. We remind that the airplane is composed of $N = 23\,962$ unknowns, 24157 vertices and 48119 elements.The simulation has to perform 10 823 time iterations, and there are $K^{max} = 341$ interaction matrices. The total number of non-zero values in the interaction matrices, except M^0, is 5.5×10^9. The longest row-vector d_{max} is 15 and the row-vectors have 9.5 values in average. For one iteration, the total amount of Flop to compute the summation s^n is about 11 $GFlop$. If we consider that the solution step of the system associated with M^0 has the cost of a matrix-vector product, the total amount of Flop for the entire simulation is 130 651 $GFlop$. In single precision we need 50 GB to store all the data of the simulation such that we need at least 4 $SSE - Configuration$ nodes to have the entire test case fitting in main memory.

Number of Blocks from the Full-Blocking

The Full-Blocking generates blocks from a given slice depending on the block size (b_r and b_c) and the position of the NNZ values inside the slices. Therefore, the numbering of the unknowns is crucial to decrease the number of blocks. We tested different ordering methods, but the spatial ordering gave better results (the comparison is not given in the current study). For example, we tried to number the unknowns by solving a Traveling Salesman Problem, using one or several interaction matrices as proposed in [22]. Here we present the results for the Morton indexing [74] and the Hilbert indexing [75]. In both cases, we compute a unique index for each unknown and sort them accordingly. It is possible to score the quality of the ordering by looking at the contiguous values between the row-vectors. If two consecutive row-vectors v_i^j and v_{i+1}^j from $Slice^j$ have $q_{i,i+1}^j$ values on the same columns, we describe the quality between these two rows as $(q_{i,i+1}^j)^2$. The quality of a slice is the average quality between its rows and the quality of the entire system is the average quality of the slices. Using the original ordering provided by the mesh generation, the ordering score for all the slices is 33. By ordering the unknowns using Morton and Hilbert indexing, we obtain the scores 54 and 55, respectively. This means that using these spatial ordering increases the quality and makes most of the consecutive row-vectors having contiguous NNZ values in the same columns.

Table 3.8 shows the number of blocks depending on the type of ordering and the size of the blocks when we process the slices of the test case using the Full-Blocking from Section 3.4.2. It is clear that numbering the unknowns with a space-filling curve drastically reduces the number of blocks (*NBB* in the table). The table also contains an estimation of the computation time (E≈) to process the generated blocks with one GPU of the $SSE/Tesla-Configuration$, using the performance measures from Table 3.4. We see that increasing the size of the blocks does not always reduce the number of blocks. For example, with the Morton Indexing the b_r parameter does not improve the filling of the blocks significantly.

These results show the limits of the Full-Blocking approach. The best estimated time E≈ which is obtained using Morton Indexing has only 14.4% of NNZ values. That means that the memory requirement is multiplied by more than 6 and that the real kernels performance is divided by 6. Moreover, in order to find out the best block size, we need to do a complete study (at least try different blocks size for a given ordering) which cannot be carried out in a real simulation. The Contiguous-Blocking approach, on the other hand, only needs to know the longest row-vector d_{max} which is 15 in the airplane test case and can be deduced from the simulation properties. With an average length of 9.5 for the row-vectors and 23 962 unknowns we have 63% of NNZ using Contiguous-Blocking. Moreover, the Contiguous-Blocking is faster on GPU than the Full-Blocking. Therefore, we do not use the Full-Blocking method in the rest of the paper to run the simulation.

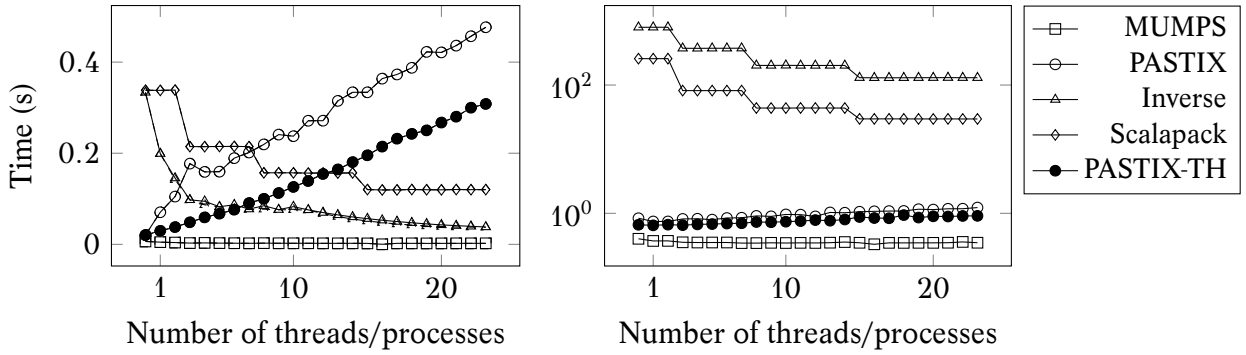
3.7.5 Linear Solvers for M^0

The choice of the linear solver is an important criterion to ensure the scalability of the whole application. Figure 3.23 shows some timing for different solvers to work with the M^0 matrix of

$b_r \times b_c$	No Ordering			Morton Indexing			Hilbert Indexing		
	NBB	NNZ%	E≈	NBB	NNZ%	E≈	NBB	NNZ%	E≈
32 x 16	$121 \cdot 10^6$	8.8%	3.1	$36 \cdot 10^6$	29.6%	0.95	$36 \cdot 10^6$	29.6%	0.95
32 x 32	$115 \cdot 10^6$	4.6%	4.6	$18 \cdot 10^6$	29.1%	0.74	$18 \cdot 10^6$	28.9%	0.75
32 x 64	$114 \cdot 10^6$	2.3%	7	$9 \cdot 10^6$	27.5%	0.59	$9 \cdot 10^6$	27.3%	0.6
32 x 128	$114 \cdot 10^6$	1.1%	13.7	$5 \cdot 10^6$	23.8%	0.68	$6 \cdot 10^6$	22.4%	0.72
64 x 16	$74 \cdot 10^6$	7.2%	**2.25**	$36 \cdot 10^6$	14.9%	1.08	$36 \cdot 10^6$	14.9%	1.08
64 x 32	$63 \cdot 10^6$	4.2%	2.75	$18 \cdot 10^6$	14.9%	0.78	$18 \cdot 10^6$	14.9%	0.78
64 x 64	$59 \cdot 10^6$	2.2%	3.82	$9 \cdot 10^6$	14.8%	0.58	$9 \cdot 10^6$	14.7%	0.58
64 x 128	$58 \cdot 10^6$	1.1%	7	$4 \cdot 10^6$	14.4%	**0.56**	$4 \cdot 10^6$	14.3%	**0.56**
128 x 16	$48 \cdot 10^6$	5.6%	2.71	$35 \cdot 10^6$	7.5%	2.02	$35 \cdot 10^6$	7.5%	2.02
128 x 32	$32 \cdot 10^6$	4.1%	2.68	$17 \cdot 10^6$	7.4%	1.47	$17 \cdot 10^6$	7.5%	1.47
128 x 64	$25 \cdot 10^6$	2.6%	3.30	$9 \cdot 10^6$	7.4%	1.15	$9 \cdot 10^6$	7.4%	1.15
128 x 128	$22 \cdot 10^6$	1.4%	5.27	$4 \cdot 10^6$	7.4%	1.06	$4 \cdot 10^6$	7.4%	1.05

Table 3.8: Number of blocks for the Full-Blocking and the airplane test case. Number of blocks generated by the Full-Blocking method in the airplane test case for different block sizes ($b_r \times b_c$) and different orderings. For each size and ordering we show the number of blocks (NBB), the percentage of non-zeros in the blocks (NNZ%) and the estimated time to compute the blocks with one GPU when using the GPU kernel performance measures (E≈) from the $SSE/Tesla-Configuration$.

the airplane test case. The factorizing time is important, but since we solve the system at each iteration the aim is to chose the solver which minimizes the total time $T_{Facto} + N_{Loop} \times T_{Solve}$. We explain in Section 3.5.3 why the usage of dense solver might be beneficial and in the study we test a dense linear solver (*Scalapack*) but we also inverse the matrix to obtain $(M^0)^{-1}$ and use the matrix-vector product during the solve (*Inverse*). We see that the *Inverse* becomes more and more competitive as the number of processes increases but the factorizing time is still expensive. For large test cases, with a small dimension and a large number of nodes, the usage of dense solver can be appropriate otherwise sparse solver are more pertinent. In the airplane test case, the sparse solvers do not improve as the number of nodes increases. In addition, we have developed and tested an iterative solver (based on the conjugate gradient method) but it appears to be slower than the ones presented here and needs 4 and 1 seconds for the solve step using 1 and 24 threads respectively.



(a) Solve time for different linear solvers

(b) Factorizing time for different linear solvers

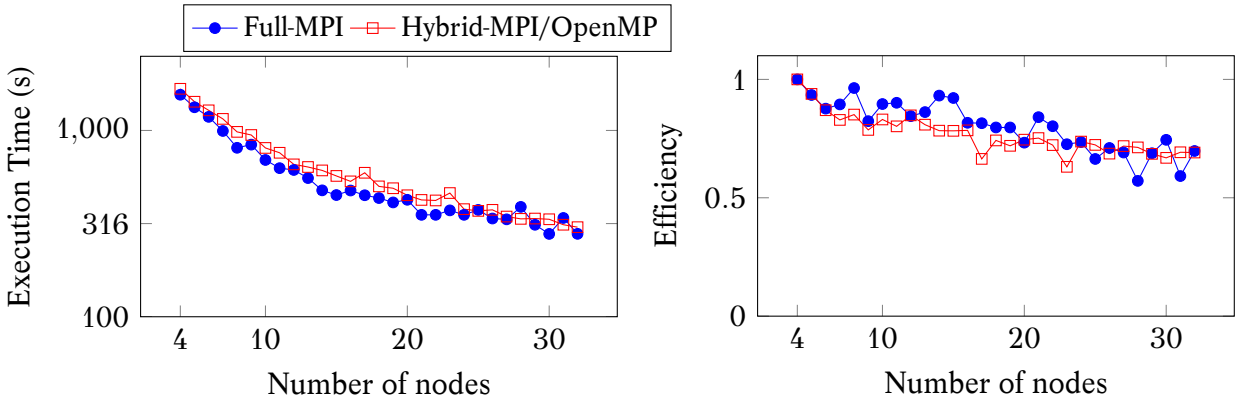
Figure 3.23: Linear solvers timing for the Airplane M^0 using the $AVX - Configuration$ CPU. We parallelize in distributed memory (MPI) except for the PASTIX-TH solvers where we use shared memory parallelization.

3.7.6 Parallel study

Homogeneous Configurations

We compare the Full-MPI and the Hybrid-MPI/OpenMP implementations to compute the airplane test case.

SSE − Configuration. We use 4 to 32 nodes and 8 cores per node. Figure 3.24 gives the total execution time and the parallel efficiency. The efficiency is worthy for both implementations but in terms of execution time, the Full-MPI is better. Even if the number of processes involved in the global communications becomes larger because there are 8 MPI processes on each node, there is no advantage in reducing this number by having one process per node and intra-node parallelism using threads. Figure 3.25 gives the percentages of time taken by the different operations. The time spent for the summation decreases as the number of node increases for both implementations. However, we see that the Hybrid-MPI/OpenMP implementation exhibits more idle time than the Full-MPI when the number of node increases. In the Hybrid-MPI/OpenMP implementation, some parts of the code are sequential; the threads share data, they parallelize small operations like the radiation for instance and the work is balanced statically between threads. In consequence, there are less MPI-processes in the Hybrid-MPI/OpenMP implementation but the threads are less balanced, and they have to wait longer in the synchronization/reduction points. Moreover, in the Hybrid-MPI/OpenMP we do not manage the NUMA effects, and our slices are allocated in a single block by the master thread and shared by all the threads of the node.

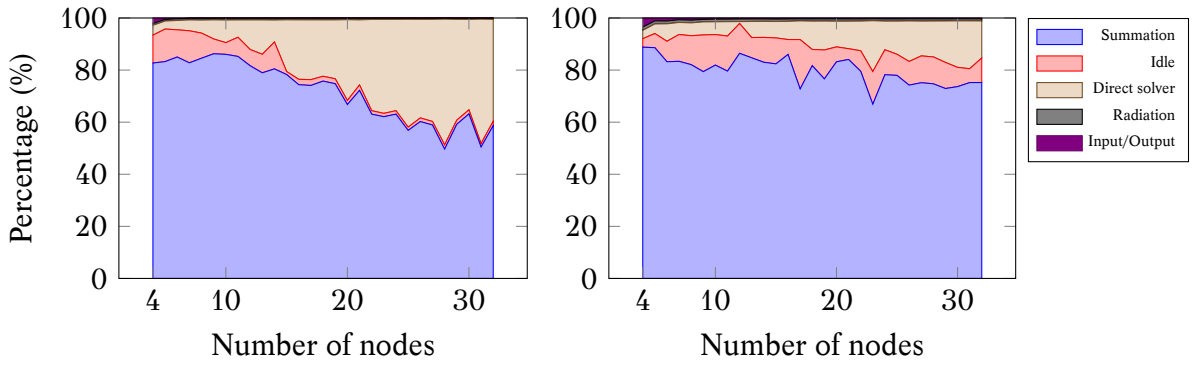


(a) Execution time

(b) Parallel efficiency $\widetilde{e}_n$

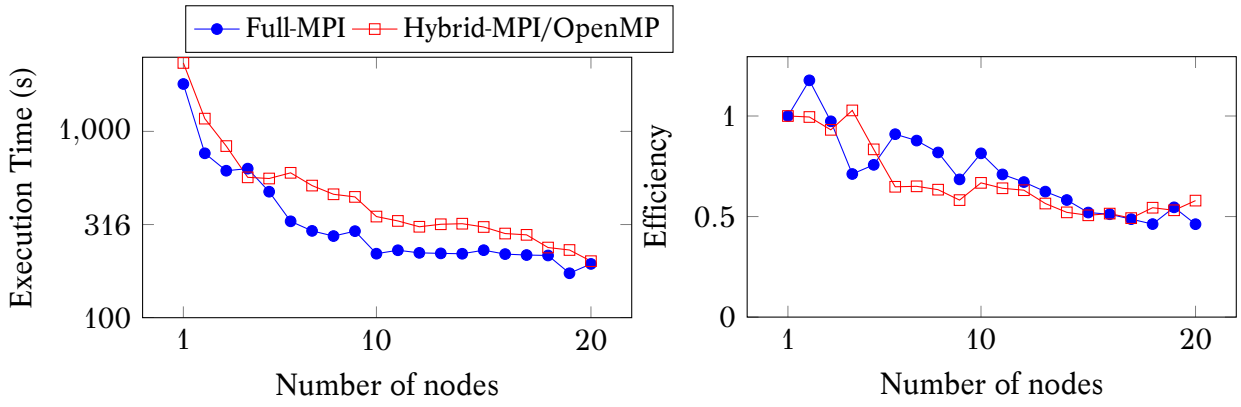
Figure 3.24: Execution time and parallel efficiency of the airplane simulation for the Full-MPI and the Hybrid-MPI/OpenMP implementations in Double using 4 to 32 nodes, 8 CPU per node and $n_g = 8$ for the *SSE − Configuration*.

AVX − Configuration. We use 1 to 20 nodes and 24 cores per node. Figure 3.26 gives the time and the efficiency and Figure 3.27 gives the percentage for each of the operations. As for the *SSE − Configuration*, the Hybrid-MPI/OpenMP implementation is slower for the same reasons. In the percentage, we see that the linear solver is becoming critical when having 20 nodes in the Full-MPI since it takes half the time.



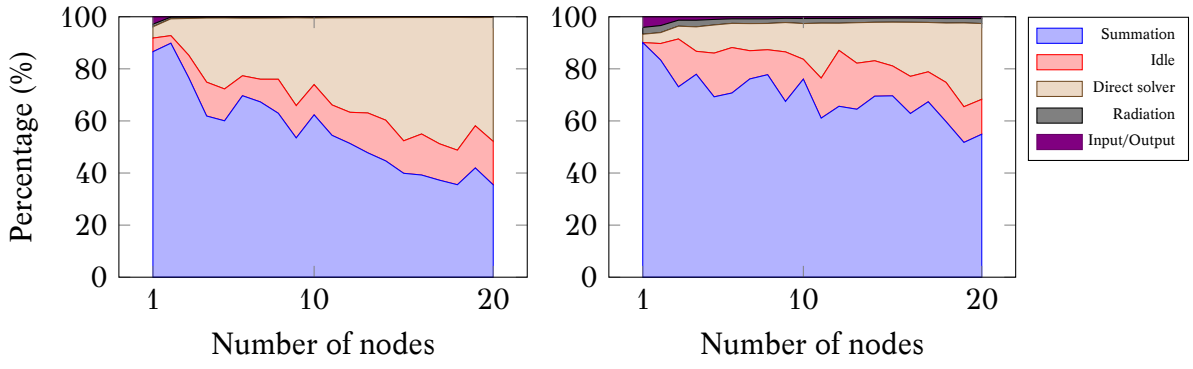
(a) Full-MPI **(b)** Hybrid-MPI/OpenMP

Figure 3.25: Percentage of the time taken for the different operations to compute the airplane simulation for the Full-MPI and the Hybrid-MPI/OpenMP implementations in Double using 4 to 32 nodes, 8 CPU per node and $n_g = 8$ for the $SSE - Configuration$.



(a) Execution time **(b)** Parallel efficiency $\widetilde{e}_n$

Figure 3.26: Execution time and parallel efficiency of the airplane simulation for the Full-MPI and the Hybrid-MPI/OpenMP implementations in Double using 1 to 20 nodes, 24 CPU per node and $n_g = 8$ for the $AVX - Configuration$.



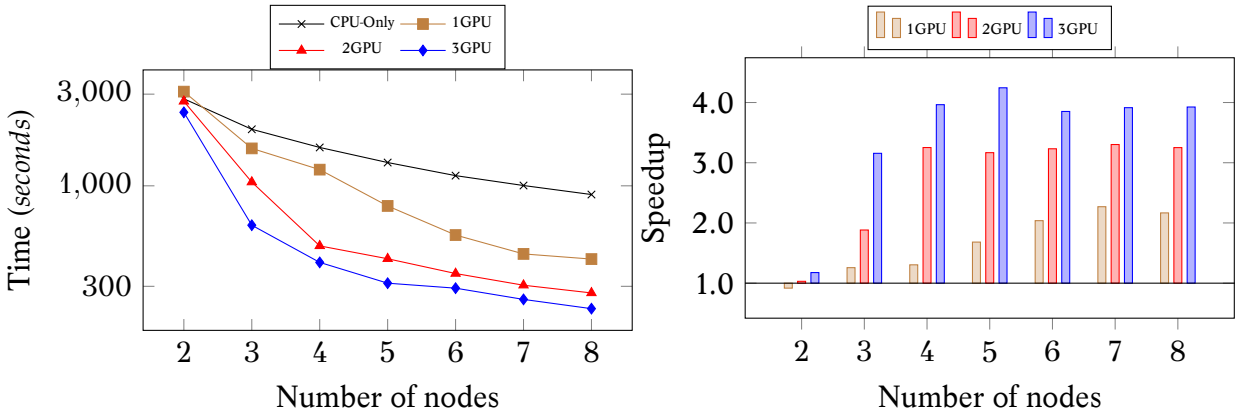
(a) Full-MPI **(b)** Hybrid-MPI/OpenMP

Figure 3.27: Percentage of the time taken for the different operations to compute the airplane simulation for the Full-MPI and the Hybrid-MPI/OpenMP implementations in Double using 1 to 20 nodes, 24 CPU per node and $n_g = 8$ for the $AVX - Configuration$.

Heterogeneous Configurations

We now study the parallel behavior of the application on heterogeneous architecture, first on the $SSE/Tesla - Configuration$ and secondly on the $AVX/Kepler - Configuration$.

SSE/Tesla — Configuration. Figure 3.28a shows the wall time to compute the simulation using 0 to 3 GPUs and 2 to 8 nodes. The data of the problem cannot be hosted by a single node and to ensure in-core execution we have to start from 2 nodes even if our application can manage out-of-core simulations. Based on these results, Figure 3.28b shows the speed-up of the GPU versions against the CPU only version. We recall that the GPU versions use all the CPUs as explained in Section 3.5.1. For a small number of nodes, the executions with GPUs do not provide a significant improvement against the CPU only case. This is because GPUs are limited by their memory capacities, and they cannot hold a large proportion of the data.Since the data are almost divided by the number of nodes, a small number of nodes means that each of them will need to store a large amount of data. When a GPU is in charge of an interval that exceeds its memory capacity, it needs to perform host-to-device copies during the computation. Such copies are slow and drastically decrease the efficiency of the GPUs. However, our application must be able to support out-of-core executions where an entire simulation cannot fit inside the main memory. It is then required to perform host-to-device or hard-drive to device copies. The balancing algorithm is responsible of the attributions of the intervals as detailed at the end of this section. The parallel efficiency of the CPU only version for 8 nodes is 0.78.



(a) Execution time

(b) Speedup against CPU-Only

Figure 3.28: Parallel study of the airplane test case from 0 to 3 GPUs and 2 to 8 nodes for the *SSE/Tesla — Configuration* in Single.

Figure 3.29 presents the details of the main operations of the simulation by giving the percentages of the operations against the complete wall time. We see that the idle time (red) remains low in most of the cases. It is clear that the solution stage becomes more and more dominant as the summation is improved. That is due to the low number of unknowns compared to the number of nodes. In fact, there is no improvement in the time spent in the solution step of the direct solver as we increase the number of nodes, and MUMPS is taking the same time with 2 or 8 nodes to solve the system for all the time steps.

We can draw the work balance between workers as shown in Figure 3.30. It shows how the work is balanced for different work distributions and different numbers of nodes in a single node for several iterations. We measure the balance by studying the time taken per each worker in terms of percentage of the total time (which is the sum of the time taken by all the workers). In

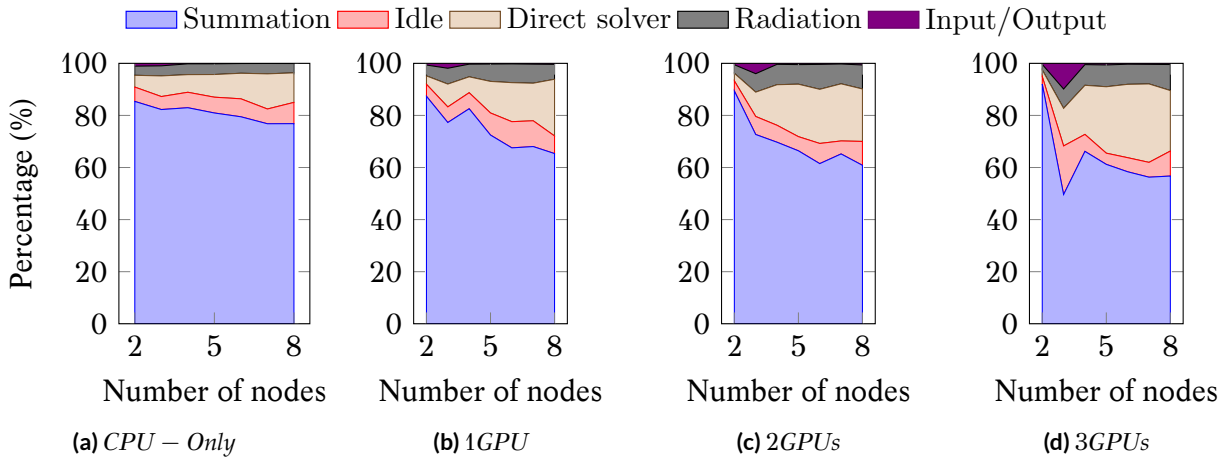


Figure 3.29: Percentage of time of the different stages of the airplane simulation for 2 to 8 nodes for the *SSE/Tesla − Configuration* in Single precision.

a perfectly balanced configuration, all the workers take $100/W$ percent of the time, with W the number of workers. We see that the balance is improved after each iteration. The second part of the figure shows the interval of work per worker. We see the speed-up of the GPU-workers against the CPU-worker when the work is balanced and what percent of the node interval each worker is in charge of. We remind that for the first iteration, the balancing algorithm gives the same number of elements to each worker. Figure 3.30a which shows the balancing for 2 nodes also points out a problem of the GPU memory limitation. In fact, at the first iteration, the GPU-worker and the CPU-worker are in charge of the same number of slices (as shown by the interval of work). However, this amount of slices cannot fit in GPU memory; thus, the first iteration is very slow for the GPU-worker and it takes 90% of the total time. The GPU-worker needs to load and copy the slices into its memory. The balancing algorithm tried to balance the second iteration, but this time the GPU-worker has few slices and can store them in its memory. Therefore, the GPU-worker computes its interval much faster than the CPU-worker. Such performance differences as in-GPU and out-of-GPU can lead to unbalanced, or at least not optimally balanced, configurations.

AVX/Kepler − Configuration. The nodes of the *AVX/Kepler − Configuration* have more GPUs and the GPUs but also more memory compared to the *SSE/Tesla − Configuration* nodes. The complete simulation can be hosted by a single node, and this is why the results start from 1 node up to 5 nodes. Again, in Figure 3.32, we see clearly the memory limit of the GPU bound the speed-up. With a single GPU per node, we need 5 nodes to have enough GPU memory and to have a complete speed-up. While with 4 GPUs the problem fit in their memory from 2 nodes. We notice that the executions using 3 nodes and 3 or 4 GPUs have important IO operations such that the summation stage appears smaller but we have to keep in mind that these results represent the percentage of time (and not the absolute time). From the percentage results, Figure 3.32, we see that the idle time remains correct.

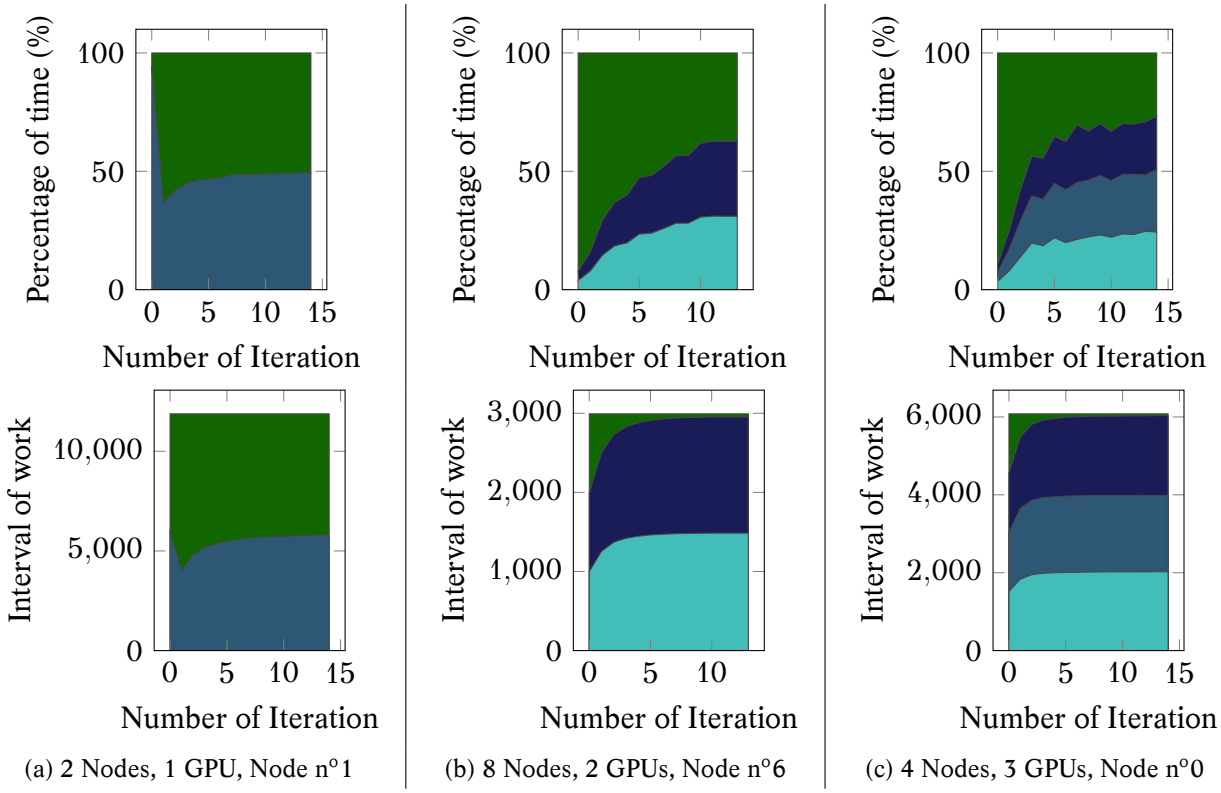
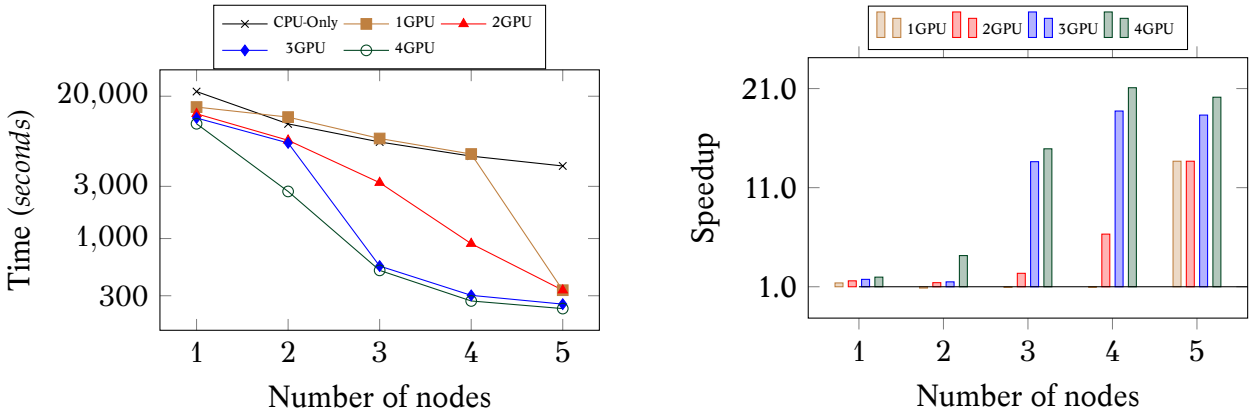


Figure 3.30: Illustration of the work balance for the airplane simulation and three hardware configurations. The goal is to have equal percentage of time per worker. The CPU-worker is always represented by green color (always at the top of the plots) and GPU-workers are plotted in blue. The balancing algorithm stops after 15 iterations. The given examples illustrate the behavior of specific nodes but the results are the same on all the nodes.



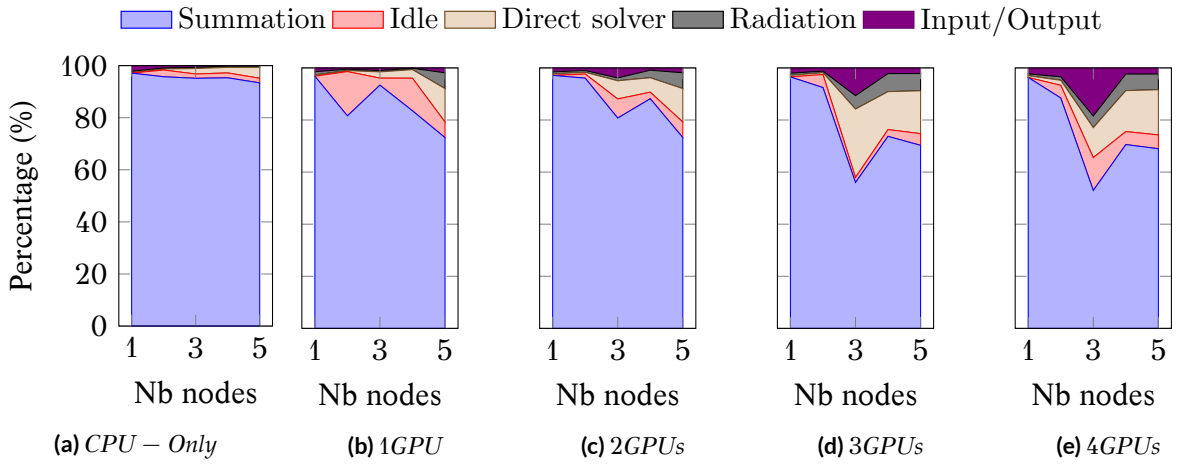
(a) Execution time

(b) Speedup against CPU-Only

Figure 3.31: Parallel study of the airplane test case from 0 to 4 GPUs and 1 to 5 nodes for the $AVX/Kepler - Configuration$ in Double.

3.7.7 Out-of-core Executions

Our application is able to compute simulations which do not fit in the main memory. However, the underlying technique has not been optimized; we simply store the slices on the hard-drive memory and create S/M groups, where S is the complete size of the slices and M the available memory. During the summation stage, we load each group but one at a time in the main memory without any allocation (a buffer of the size of the largest group is allocated and reuse). It would be possible

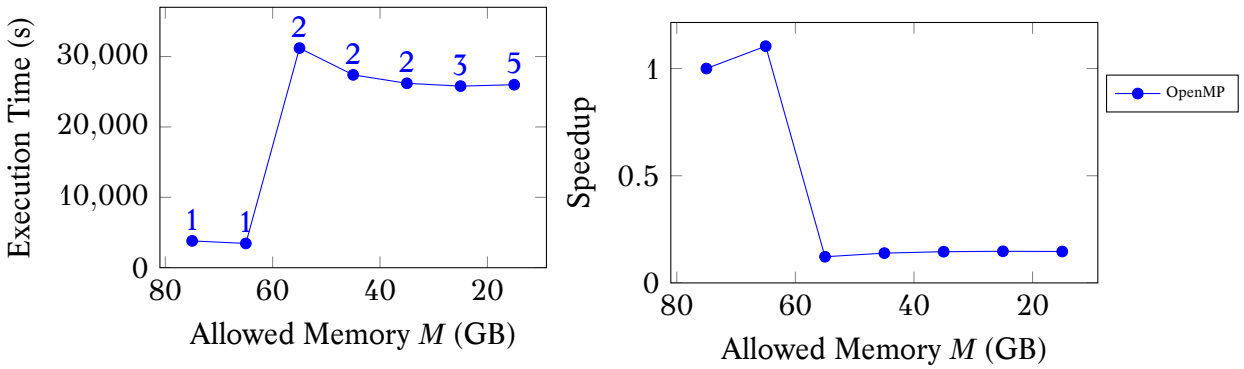


(a) $CPU - Only$ **(b)** $1GPU$ **(c)** $2GPUs$ **(d)** $3GPUs$ **(e)** $4GPUs$

Figure 3.32: Percentage of time of the different stages of the airplane simulation for 1 to 5 nodes for the $AVX/Kepler - Configuration$ in Double precision.

to improve the method by allocating two buffers and by loading and computing concurrently (which leads to $S/M/2$ groups).

In Figure 3.33, we show the execution time for different buffer-memory sizes on a cone-sphere test case. We see that once the slices does not fit in memory, which means that we have more than one group, the execution time slow-down drastically. When the allowed memory is fixed to $M = 55GB$, we have two blocks; one blocks of size $55GB$ and a second one of size $2GB$. Whereas, if the limit is fixed to $M = 35GB$, we have one block of size $35GB$ a second one of size $22GB$. From the results, it seems that the smaller the blocks, and the read from the file, the better are the performance. Even so, out-of-core capacity opens the possibility to work on small computation nodes or with large simulations.



(a) Execution time, with number of memory partitions in label

(b) Speedup (T_{75}/T_M)

Figure 3.33: Execution time and efficiency for out-of-core cone-sphere simulations for OpenMP implementations in Double using 1 node with 24 CPU and $n_g = 8$ for the $AVX - Configuration$. The cone-sphere test case C-22468 introduced in Section 1.2.4, which has 9953 time steps and needs a total memory of $57GB$ with $RHS = 2$.

3.8 Matrix Computation Summary

This chapter describe how our TD-BEM formulation can be efficiently implemented on CPU and GPU thanks to the reordering of the computation. We show that the we can achieve a high flop-rate

using advanced optimization on the different architectures and that parallelization is efficient even if it is more and more limited by the underlying linear solver. The benefit of the GPU is proved only when the data fit in the GPUs' memory which happens with a low ratio of the problem size against the number of nodes/GPUs. However, the resulting application is able to work on most hardware configurations composed by CPU/GPU thanks to the balancing heuristic.

4

Parallel Fast Multipole Method

This chapter describes the different parallelization strategies that have been developed for the generic Fast Multipole Method (FMM). It includes straightforward shared memory approaches using *fork-join* or *tasks-and-wait* models, which are also used as a basis for the hybrid parallelization. In addition, we introduce the data structure for the *tasks-and-dependencies* FMM in shared and distributed memory with the support of a runtime system. For the entire chapter, we consider the use of Morton indexing and restrict our definition to three dimensions ($3D$). The related developments have been incorporated into the ScalFMM library presented in Section 2.3.5.

In Section 2.3.3, we define the FMM algorithm as a method which reduces the number of interactions using an appropriate mathematical kernel to approximate the far-field. Many FMM-based scientific applications mix the numerical formulations and the FMM algorithm without a straight border between both layers. However, interleaving the physical problem, the mathematics and the FMM, which includes the algorithm, the data structures and some parallelization schemes, leads to complex codes. That is why, we have decided to work on the development and the parallelization of the FMM as a generic method and an external module even if our objective is to implement an FMM kernel for our TD-BEM problem. As a result, the current chapter includes a study of the FMM algorithm independently of the TD-BEM formulation.

4.1 Sequential Fast Multipole Method

4.1.1 Algorithm

In Section 2.3 we introduce the FMM and its relative operators. We also describe how the FMM algorithm reduces the complexity by an approximation of the far-field using an appropriate mathematical kernel. The FMM algorithm is expressed intuitively if we have an octree data structure which supports the following operations:

- from a cell c or a leaf lf, the octree proposes a method to get their corresponding Morton

indexes, and the other way around, with a low complexity ;

- it has an efficient linear access over the cells; for a given level l, the octree provides a method to iterate on all the cells of the same level in ascending Morton index order ;
- it has an efficient linear access over all the leaves, as it is for the cells, the octree provides a method to iterate on the leaves in ascending Morton index ;
- from a cell c or its Morton index and the corresponding level, the octree proposes a fast access to the children and the cells that composed the interaction list ;
- from a leaf lf or its Morton index, the octree provides access to the corresponding direct neighbors.

The indirection octree from Section 2.3.5 provides all these methods. Moreover, this data structure manages the empty space/cells, and iterating inside a level of a sparse octree is done with a low complexity. Using a such data structure allows to write the FMM as in Algorithm 8. The main function (FMM) calls the operators in the correct order, and the algorithm explicitly shows the division between the near and the far fields. The proposed algorithm computes first the near-field ($P2P$) and then on the far-field interactions, but the reverse order would have been valid too. The $M2L$ is done at level l, then the data are moved downward by the $L2L$ from l to $l+1$, and later the $M2L$ at level $l+1$ occurs and so on. It is also valid to perform all the $M2L$ at all levels, and then to do the downward pass from level 2 to $h-1$.

In our case, the $P2M$, the $M2M$, the $L2L$ and the $M2L$ compute their full interactions. For example, the $M2M$ takes all the children of a cell to compute the aggregation in the parent, but it is possible to dissociate these calls and to perform the computation between one parent and its child at a time. These choices are a matter of underlying optimizations and parallelism as explained in further sections.

4.2 Shared Memory Parallelization

In this section, we introduce the parallelization schemes based on the *fork-join* model. These algorithms can be implemented with OpenMP 3.1. We present formulations based on the OpenMP standard, but we do not strictly follows the standard in order to make the algorithms more intuitive.

4.2.1 *parallel-for*

In our sequential algorithm (Algorithm 8), the operators work with one cell/leaf at each call. This sequential implementation is straightforward to parallelize using the *parallel-for* statement. The loops are divided among the threads, and there is no possible race condition or memory conflict between the threads as long as the kernel functions carefully manage the parallel accesses. Algorithm 9 shows the parallelization for the $M2L$ operator with a simple *parallel-for* but this statement should be parametrized. In fact, we have to choose how to divide the work between the threads

Algorithm 8: FMM Sequential Algorithm

```
function FMM(tree, kernel)
    // Near-field
    P2P(tree, kernel);
    // Far-field
    P2M(tree, kernel);
    for l = tree.height-2 → 2 do
        M2M(tree, kernel, l);
    for l = 2 → tree.height-2 do
        M2L(tree, kernel, l);
        L2L(tree, kernel, l);
    M2L(tree, kernel, tree.height-1);
    L2P(tree, kernel);
```

```
function P2P(tree, kernel)
    foreach leaf lf in tree.leaves do
        kernel.P2P(lf.targets, lf.sources, tree.getSourceNeighbors(lf.mindex));
```

```
function P2M(tree, kernel)
    foreach leaf lf in tree.leaves do
        kernel.P2M(lf.sources, tree.getCell(lf.mindex).multipole);
```

```
function M2M(tree, kernel, level)
    foreach cell cl in tree.cells[level] do
        kernel.M2M(cl.multipole, tree.getChildren(cl.mindex, level).multipole);
```

```
function M2L(tree, kernel, level)
    foreach cell cl in level.cells do
        kernel.M2L(cl.local, tree.getInteractions(cl.mindex, level).multipole);
```

```
function L2L(tree, kernel, level)
    foreach cell cl in tree.cells[level] do
        kernel.L2L(cl.local, tree.getChildren(cl.mindex, level).local);
```

```
function L2P(tree, kernel)
    foreach leaf lf in tree.leaves do
        kernel.L2P(tree.getCell(lf.mindex).local, lf.targets);
```

by selecting the appropriate granularities/chunk sizes and schedule (static/dynamic). A dynamic schedule involves more critical sections, but a static division may lead to unbalanced execution if the workload is not uniformly distributed. In addition, it is possible to work on the $M2L$ and the $L2L$ separately, which allows to remove the barriers between the different level for the $M2L$.

Algorithm 9: FMM Operator using *parallel-for* (M2L)

```
function M2L_parfor(tree, kernel, level)
    #pragma omp parallel for
    foreach cell cl in level.cells do
        kernel.M2L(cl.local, tree.getInteractions(cl.mindex, level, Multipole));
    // Implicit barrier from omp parallel
```

The sequential $P2P$ operates on the leaves and their neighbors. If we parallelize this loop, the threads may access concurrently the same leaves while the $P2P$ modifies both the target cell

and it neighbors. This happens if the computational kernel is anti-symmetric such that for two particles i and j we have $F_{i,j} = -F_{j,i}$. In fact, to reduce the computational cost, the $P2P$ computes only half of the interactions and applies the results to the targets and the sources. We can use several mutexes or a lock-free mechanism to avoid the memory conflicts between the threads. However, we use an alternative solution which is known as the spacial decomposition color (or coloring pattern). We divide the original problem into sub-problems (colors) where no protection is needed inside these sub-problems (some barriers must be used between the colors). In our $P2P$, a color represents the leaves that are separated by two other leaves. As an example, in one dimension, the colors are the sequence *red green blue red green blue*... such that with a parallel loop over one color, the threads modify the leaves (*red*) and their direct neighbors (*green*, *blue*) without overlapping. In $1D$ the color is found from the grid index $lf.x$ by $lf.x \, MOD 3$, and in $3D$ by $((lf.x \, MOD \, 3) \times 3 + (lf.y \, MOD \, 3)) \times 3 + (lf.z \, MOD \, 3)$. The parallelization of the $P2P$ using this pattern is shown in Algorithm 10. It is possible to use less than 3^D colors if we consider that the $P2P$ modifies always the same neighbors relatively to the target. In $3D$, we can use 18 colors instead of 27, see [99], if the modified neighbors have always lower tree coordinates than the target leaf ($x \times 9 + y \times 3 + z < 14$).

Algorithm 10: FMM Parallel P2P with color scheme

```
function P2P(tree, kernel)
    foreach Color color in tree.leaves.criticalColors do
        #pragma omp parallel for
        foreach leaf lf in tree.leaves[color] do
            kernel.P2P(lf.sources_targets, tree.getNeighbors(lf.mindex));
```

4.2.2 *tasks-and-wait*

The difference between *tasks-and-wait* and *parallel-for* models is thin: a *parallel-for* with a dynamic scheduling or a *parallel-for* with static scheduling and a chunk size of N/p give the same parallel behavior as a *tasks-and-wait* approach. However, the *tasks-and-wait* allows for finer synchronizations because only one thread waits for the tasks to finish while the other ones continue their execution. In Algorithm 11, we give an example of parallelization where the $M2L$ is based on the *tasks-and-wait* but where the other operators still rely on the *parallel-for*. The algorithm also provides an equivalent of the $M2L\_task$ using *parallel-for* referenced as $M2L\_full\_parfor$ with the *nowait* keyword to remove the barrier after the divided loop. The granularity can be parametrized in both approaches but there are still a lot of barriers between the operators and the levels. For example, the particles are only used by the $P2P$ and the $L2P$ but no threads are able to start the $P2M$ while all the others have not finished to work inside the $P2P$.

Algorithm 11: FMM *tasks-and-wait* Algorithm and comparison between M2L_task and M2L_full_parfor.

```
function FMM(tree, kernel)
    // Near-field
    P2P_parfor(tree, kernel);
    // Far-field
    P2M_parfor(tree, kernel);
    for l = tree.height-2 → 2 do
        M2M_parfor(tree, kernel, l);
    M2L_task(tree, kernel);
    for l = 2 → tree.height-2 do
        L2L_parfor(tree, kernel, l);
    L2P_parfor(tree, kernel);
```

```
function M2L_task(tree, kernel)
    #pragma omp parallel
        #pragma omp single
            for l = 2 → tree.height-2 do
                foreach cell cl in level.cells do
                    #pragma omp task
                    kernel.M2L(cl.local, tree.getInteractions(cl.mindex, level,
                    Multipole));
        // Implicit end of the tasks
```

```
function M2L_full_parfor(tree, kernel)
    #pragma omp parallel
        for l = 2 → tree.height-2 do
            #pragma omp for schedule(dynamic,1) nowait
                foreach cell cl in level.cells do
                    kernel.M2L(cl.local, tree.getInteractions(cl.mindex, level,
                    Multipole));
                // No barrier, the threads go to the next loop (nowait)
        // Implicit wait
```

4.2.3 Section *tasks-and-wait*

The near and far fields are independent, and we can compute them concurrently (except for the *L2P* and the *P2P* because they both update the particles). Using the *tasks-and-wait* paradigm, we can divide the FMM in two independent execution paths as written in Algorithm 12. One thread inserts the task for the near-field and ensures the coherency using barriers where required, while another thread inserts the tasks for the far-field and ensures the correct execution using barriers. The other threads pick tasks from the near-field or the far-field, and we expect to reduce the idle time until one of the two sequences is over. At the end, the *L2P* is used as a reduction operation between the near and far fields, but it must start only when both previous sections are over.

The main drawback of this implementation comes from the lack of priority system in the OpenMP standard [*]. Therefore, the execution is tied to the underlying OpenMP implementation (the tasks may be stored in FIFO or LIFO fashion). There are still a lot of synchronizations inside the se-

[*]The OpenMP standard will introduce priorities in its revision 4.5.

quences, and it would certainly be a benefit to have finer dependencies. For example, in the $M2M$ from Algorithm 12, the *taskwait* ensures that the work at level $l+1$ must be over to work at level l. This level-to-level dependency can be replaced by dependencies between cells such we compute the $M2M$ between a cell and its children as soon as the children are ready. Such approaches are presented in Section 4.4.

Algorithm 12: FMM Section *tasks-and-wait* Algorithm

```
function FMM(tree, kernel)
    #pragma omp parallel
        #pragma omp single nowait
            // Near-field
            P2P_task(tree, kernel);
        #pragma omp single nowait
            // Far-field
            P2M_task(tree, kernel);
            for l = tree.height-2 → 2 do
                M2M_task(tree, kernel, l);
            M2L_task(tree, kernel);
            for l = 2 → tree.height-2 do
                L2L_task(tree, kernel, l);
    // Merge
    L2P_parfor(tree, kernel);
```

```
function M2M_task(tree, kernel)
    foreach cell cl in tree.cells[level] do
        #pragma omp task kernel.M2M(cl.multipole, tree.getChildren(cl.mindex,
        level).multipole);
    #pragma omp taskwait
```

4.3 Hybrid Parallelization (MPI/OpenMP)

4.3.1 Distributed Memory (Full-MPI)

The communication between the computational nodes is done using the Message Passing Interface (MPI), and we use the term *process* to name an MPI process. In MPI, each process has an individual rank, and we use the terms left and right processes relatively to process k to denote the processes that have lower or greater ranks than k.

In our distributed implementation, we do not duplicate the entire octree over the nodes; for large test cases, a complete octree may not fit into the memory of a single node. On the other hand, we have to divide our octree but we also need to keep a good data locality inside each node and a low communication cost between the nodes. In Section 2.3.4, we show that the Morton and Hilbert indexing have good properties for a dense octree. More precisely, the model study the bottom $M2L$ and shows that the number of messages, the size of the messages and the locality are much better compared to a linear indexing. The simplicity to compute the Morton indexing, compared to the Hilbert indexing, and its intrinsic hierarchical structure make it very convenient and commonly used in the FMM.

To distribute the octree, we divide the global Morton interval at the leaf level (h_l) between all the processes: each process p_i is in charge of all the particles included between the Morton indexes $M_i^S(h_l)$ and $M_i^E(h_l)$. The particles are distributed in an increasing order for both the process rank and the Morton index, and for $p_i < p_j$ we have $M_i^S(h_l) \leq M_i^E(h_l) < M_j^S(h_l) \leq M_j^E(h_l)$. This relation constrains the leaves to be unique; no particles with the same Morton index are hosted by different processes, and in the case of a dense octree we have $M_i^E(h_l) = M_{i+1}^S(h_l) - 1$, with $i < np - 1$ where np is the number of processes. The limited memory constrains the distribution of the particles and the construction of the octree which are done in four steps.

In the first stage, each process loads N/np of the total simulation particles and computes their corresponding Morton indexes. Then, the particles are sorted among all the processes using a parallel sort (without merging on a single node). In our current implementation, we use a parallel Quicksort as presented in Appendix D.3. After these stages, the particles are sorted in Morton indexing order among all the processes but some leaves are potentially split over several processes and the distribution is possibly strongly unbalanced due to the sorting algorithm. The next steps tackle these problems, all the processes communicate to check at the extremities of their neighboring processes and merge the leaves appropriately. During the last step, the processes balance the leaves using a chosen strategy as for example assigning to each process the same number of particles or the same number of leaves. After this loading procedure, the processes build their individual local octree over their distinct intervals $M_i^S(h_l) \leq M_i^E(h_l)$.

A local octree is similar to the one used in shared memory parallelization: it is an octree by indirection with the particles hosted in the leaves and with the cells created from the leaves to the root over non empty areas. While the leaves are unique among all the processes, this is not the case for the cells above (positioned at levels $< h_l$). This is illustrated in Figure 4.1, the parents of the leaves and the upper cells may exist on several processes and for example the root exists on all the processes. These duplicate cells are called the join-cells because they are located on the extremities/borders of the Morton intervals of the processes. They should not be confused with the ghost cells that are located on the spatial boundary between the cells hosted by the different processes. This extends the previous relation for upper levels by $M_i^S(l) \leq M_i^E(l) \leq M_j^S(l) \leq M_j^E(l)$ for $i < j$ and $l < h_l$. Moreover, we know the Morton interval of the upper level of a process by performing a bit shift operation: at level l process p_i covers from $M_i^S(l) = \mathit{bits\_right\_shift}(M_i^S(h_l), D \times (h_l - l))$ to $M_i^E(l) = \mathit{bits\_right\_shift}(M_i^E(h_l), D \times (h_l - l))$.

We do not duplicate the octree, and we do not construct an octree representation which tells which cells exist on each process because we consider that it is not needed if the octree is divided following the global index. However, each process knows the working intervals of the others at the leaf level. From this information, the processes only know what others are covering but they do not know which cells exist or not inside these intervals.

The join-cells exist in the octrees of several nodes, and we decide that it is the process with the lowest rank among the owners of a join-cell that manages it. By looking again at Figure 4.1, P^0 is responsible of A and B and P^1 is responsible of C. Therefore, for the upper levels this defines a difference between the covering interval $[M^S; M^E]$ (the existing cells) and the working interval

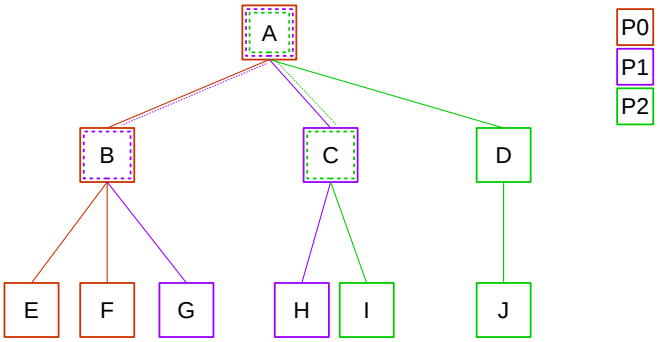


Figure 4.1: Distributed Octree among 3 processes each with two leaves and with the join-cells: A, B and C. Covering intervals include: $M_0 = \{\{A\},\{B\},\{E,F\}\}, M_1 = \{\{A\},\{B,C\},\{G,H\}\}, M_2 = \{\{A\},\{C,D\},\{I,J\}\}$. Working intervals include: $W_0 = \{\{A\},\{B\},\{E,F\}\}, W_1 = \{\{\},\{C\},\{G,H\}\}, W_2 = \{\{\},\{D\},\{I,J\}\}$.

$[W^S; W^E]$ (the cells to manage). The processes are responsible for the cells inside their working intervals and thus they may need to skip the join-cells that are inside their covering intervals. The computation of the working interval is presented in Algorithm 13 where the beginning of an interval is the maximum of the covering interval and the end of the covering interval of the left process.

Algorithm 13: Working interval per process with output W^S ($W[:][:][0]$) and W^E ($W[:][:][1]$). If a process p has $W[l][p][0] > W[l][p][1]$ then it does not have work to do at level l and above.

Data: h the height of the tree, np the number of processes
function `FindWorkingIntervals(Morton Index myInterval[2]) : W[h][np][2]`
- `// All exchange their intervals at the leaf level (`h_l`)`
- $W^{\cdot}_{:}(h_l)$ = send_receive_bottom_interval(myInterval);
- **for** `l = h-2` $\rightarrow$ `0` **do**
  - `// Compute for p0 at level l`
  - $W^S_0(l)$ = bits_right_shift($W^S_0(l+1)$, 3);
  - $W^E_0(l)$ = bits_right_shift($W^E_0(l+1)$, 3);
  - `// Compute for others with their left process as limit`
  - **for** `process = 1` $\rightarrow$ `np-1` **do**
    - $W^S_{process}(l)$ = Max(bits_right_shift($W^S_{process}(l+1)$, 3), $W^E_{process-1}(l)$+1);
    - $W^E_{process}(l)$ = bits_right_shift($W^E_{process}(l+1)$, 3);

We present the distributed operators between level, which includes the $P2M$, the $M2M$, the $L2L$ and the $L2P$, and the transfer operators ($P2P$ and $M2L$).

Transfer and direct passes. We process the $P2P$ and the $M2L$ in the similarly because they both modify local data and need remote data. In the current description, we introduce the algorithm for the $M2L$ but it also applies to the $P2P$ by reducing the interaction list of the leaves.

The $M2L$ operator is computed in two distinct steps categorized as *local* and *distributed*. In the local step, each process computes the $M2L$ between the cells it hosts without any communication. Thanks to the Morton indexing we expect that only few interactions close to the boundaries are missing after this stage.

In the distributed step, the processes find the potential missing interactions, exchange the corresponding cells and complete the $M2L$. To find the missing interactions, a process iterates on its local cells and computes their interaction lists where each list is an array of 189 Morton indexes as

in the usual FMM. Then, for each of these indexes, the process finds the corresponding working interval and the potential owner of the remote cell. If an index is not included in any interval then it guarantees that the corresponding cell does not exist. The process sends all the cells which have at least one remote interaction to the remote cell owners, and it expects to receive the remote cells from them. For example, let have the process p_i which computes the Morton indexes of the interaction list I of the cell cl. By iterating on I, it finds that one of this indexes m is not inside its working interval but is in the working interval of the process p_j. It means that p_i must send cl to p_j and mark cl as incomplete. The process p_i does not know if the remote cell of index m exists on p_j but it considers it does. If the cell of index m exists, then p_j will also find out that p_i needs this cell of index m and will proceed the same way. Once all the data have been exchanged, the distributed stage ends by the computation of the operators between the marked cells and the received ones.

The amount of exchanged data is driven by the quality of the indexing, and we expect it to be low thanks to the Morton indexing qualities. However, if a process has N_{leaves} leaves and N_{cells} cells, then the number of communications is bounded by $26 \times N_{leaves}$ and $189 \times N_{cells}$ for the $P2P$ and the $M2L$ respectively. In addition, if the length of the working interval is $W_i(l) = W_i^E(l) - W_i^S(l) + 1$, the number of received is bounded by $26 \times W_i(l)$ and $189 \times W_i(l)$ for the $P2P$ ($l = h_l$) and the $M2L$ ($l \leq h_l$).

Upward and downward passes. The $P2M$ and $L2P$ are done without any communications because the leaves and the corresponding cells at the leaf level are hosted by the same processes. However, the $M2M$ and the $L2L$ require communications from the reduction of the covering intervals to the working intervals; as shown in Figure 4.1, during the $M2M$ the process $P1$ sends G to the process $P0$ and the process $P2$ sends I to the process $P1$ while the inverse is necessary during the $L2L$, $P0$ sends B to $P1$ and $P1$ sends C to $P2$. The $M2M$ and the $L2L$ can be described in two distinct stages which are applied level per level.

In the local $M2M/L2L$ a process performs the work between the cells in its working interval and their children. However, before moving to the next level, it may send or receive cells at the borders which is done during the distributed step. We remind that a process only knows the working and covering intervals of the others, and using this information we define three functions:

- *proc_has_work*(*level* l, *process* p): tells if a process p is in charge of at least one cell at level l. The function returns true if $W_p^E(l) - W_p^S(l) \geq 1$. If a process does not have work to do at level l then it is also true for all the upper level $< l$

- *proc_covers_right*(*process* r, *level* l, *process* p): tells if a process p has a join-cell with process r ($r < p$) i.e. if p has at least one child of the last cell of the working interval of process r at level l. This function returns true if $bits\_right\_shift(W_r^E(l+1), D) == bits\_right\_shift(W_p^S(l+1), D)$

- *proc_covers_left*(*process* r, *level* l, *process* p): tells if a process p has a join-cell with process r ($p < r$) i.e. if r has at least one child of the last cell of p. This function returns true if $bits\_right\_shift(W_p^E(l+1), D) == bits\_right\_shift(W_r^S(l+1), D)$.

Using these functions, the $M2M$ is written as in Algorithm 14. Each process knows what it has to send or to receive by looking at the working and covering intervals of the others in the current and the lower levels. The communications are bounded and during the upward pass at a given level l, one process p_r sends a maximum of 7 cells from level $l+1$ to only to one process p_j. This also means that if there exists a process x with $j < x < r$ this one does not have work to do at level l and above. From the receiver point of view, a process r obtains a maximum of 7 cells from a maximum of 7 distinct processes. Except for the processes at the extremities with ranks 0 and $np-1$, any process might send a cell at level l if it has work at level $l+1$ and receives if it has work at level l.

Algorithm 14: Send/Receive in distributed $M2M$ for all levels.

```
Data: h the height of the tree
function M2M_comm()
    idx_send_to = my_rank-1;
    first_idx_recv = my_rank+1;
    for l = h-2 → 2 do
        if proc_has_work(l+1, my_rank) == FALSE then
            // No possible contributions
            break;
        // Find first left process with work at current level
        while idx_send_to ≥ 0 AND proc_has_work(l, idx_send_to) == FALSE do
            idx_send_to = idx_send_to - 1;
        if idx_send_to ≥ 0 AND proc_covers_left(my_rank, l, idx_send_to) then
            send first cells of level l + 1 that have W^E_{idx_send_to}(l) as parent and which are inside my_rank working
            interval W^S_{my_rank}(l + 1) to process idx_send_to
        // Find first right process with work at lower level
        while first_idx_recv < np AND proc_has_work(l+1, first_idx_recv) == FALSE do
            first_idx_recv = first_idx_recv + 1;
        if first_idx_recv < np AND proc_covers_right(my_rank, l, first_idx_recv) then
            receive from first_idx_recv at most 7 children of W^E_{my_rank}(l + 1) while first_idx_recv < np AND
            proc_has_work(l, first_idx_recv) == FALSE do
                first_idx_recv = first_idx_recv + 1;
                if first_idx_recv < np AND proc_has_work(l+1, first_idx_recv) AND proc_covers_right(my_rank,
                l, first_idx_recv) then
                    receive from first_idx_recv at most 7 children of W^E_{my_rank}(l + 1)
```

A similar approach is used for the $L2L$ but this time the emitters become receivers and the receivers become emitters. In addition, each process sends a maximum of one cell (the join-cell) but potentially to 7 processes which include all the processes that have at least a child of the join-cell inside their working interval at level $l+1$.

4.3.2 Communication Hiding Strategy

The only difference between the shared memory strategies, presented in Section 4.2, and the local steps of the hybrid distributed memory algorithm is that the computation is done on the working intervals and not the covering intervals. In addition to the local computation, there is a commu-

nication stage and a second computation stage which includes the remote interactions. Mixing OpenMP threads and MPI calls should be done carefully.

In our implementation, one thread is responsible for all the communications while the others are available for computation. This strategy is shown in Algorithm 15 for the $P2P$ and the $M2M$ but the $M2L$ and the $L2L$ are implemented similarly. A single thread first manages the communications while the others proceed the local computation. If the communications are over before the computation, the communication thread joins the others and participates to the work. Otherwise, all the threads meet at the barrier which is crossed once the communications and the local work are over. Finally, all the threads are involved in the remote computation using the received data. Our communication thread posts non-blocking sends and receives (MPI ISend/IRecv) and then waits for completion (MPI waitall) ensuring that once it leaves its single section, the communications are finished. With this approach, we do not interleave the near and far-field but it would have been possible with a multi-thread MPI support and using two communication threads and more memory.

Algorithm 15: FMM - Communication hiding examples

```
function P2P_distr(tree, kernel)
    #pragma omp parallel
        #pragma omp master // nowait
            // Send/Receive non-blocking with a waitall
            received_leaves = exchange_leaves(tree);
        #pragma omp single // wait
            // Local P2P
            P2P_task(tree, kernel);
        #pragma omp single
            // P2P using received data
            P2P_task_distributed(tree, kernel, received_leaves);
function M2M_distr(tree, kernel)
    for l = h-2 → 2 do
        #pragma omp parallel
            #pragma omp master // nowait
                // Send/Receive as in Algorithm 14 non-blocking with a waitall
                received_cells = exchange_cell_M2M(tree, l);
            #pragma omp single // wait
                // Local M2M
                M2M_task(tree, kernel);
            #pragma omp single
                // M2M using received data
                M2M_task_distributed(tree, kernel, received_cells);
```

4.4 Tasks-and-Dependencies Parallelization

The presented strategies in Section 4.2 suffer from a poor parallelism expression coming from extra-constraints. Having a barrier or a task-wait between two levels means that to work on any cell on a given level all the work on the previous level has to be over. It appears interesting to have

finer dependencies to relax some parallelism and study the potential gain. Moreover, a *parallel-for* is appropriate on shared memory, but it is difficult to incorporate accelerators with this type of *fork-join* approaches.

4.4.1 FMM Direct Acyclic Graph (DAG)

To simplify our description, we use the term DAG to represents the tasks and their dependencies even if it is not a direct acyclic graph but more a extended graph with advanced features. In Figure 4.2 we present a DAG for a given FMM problem, but it is not an execution DAG because it includes commutative operations that cannot be symbolized with a direct acyclic graph. In the FMM most operators are commutative because they represent an aggregation of information (*plus-equal*), and the given graph explicitly show these relations. During the execution of this graph, different choices must be done to decide in which order to tasks are computed.

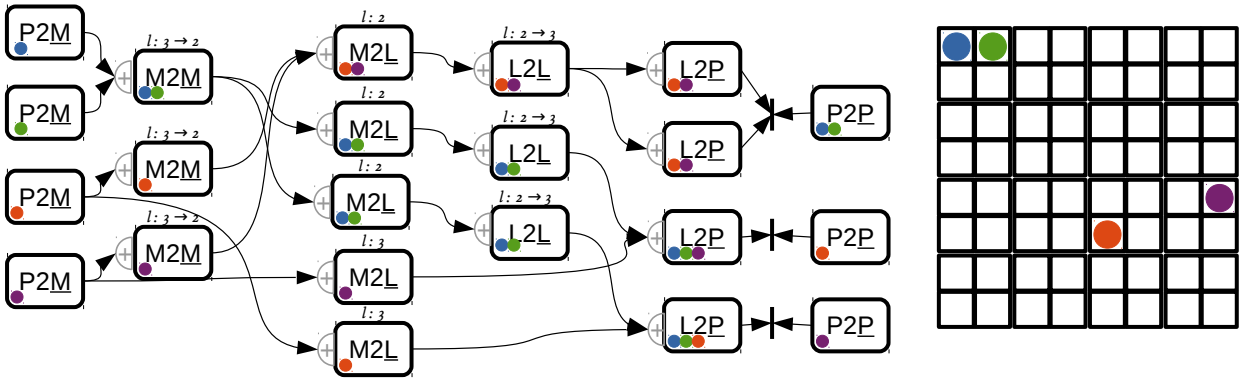
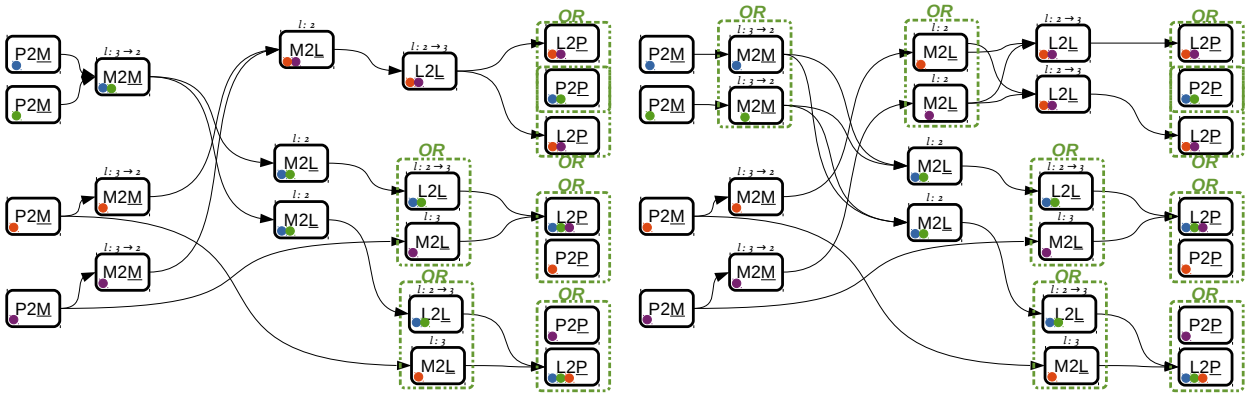


Figure 4.2: FMM pseudo-DAG in $2D$ for a height $h = 4$. The circles show how the information transit from the leaves to impact all the other leaves. The $+$ symbolized the commutativity of the operation. The bar | represent commutative access to the same data.

The commutative property is clear between operators; the $M2L$ and the $L2L$ update the local parts of the cells, and the $L2P$ and the $P2P$ update the particles. However, the operators are also commutative internally. It is possible to compute the $M2M$ between a parent and its children in any order. Similarly, for the $M2L$, the interaction list can be proceed in any order, and it is even possible to compute some $M2L$ interactions, then to apply the $L2L$, and to finish by the remaining $M2L$. In the fork-join approaches, these operations are not dissociated, and a $M2M$ implies one parent and all its children for example. All these choices lead to different executions, and we present two DAGs where we dissociate or not the operators in Figure 4.3a and Figure 4.3b. If the commutativity is supported, the dissociation of the operators gives more parallelism but reduces the granularity. On the other hand, if the commutativity is not supported, dissociating the operations does not increase the parallelism because a hard order is created during the task insertion.

The management of the commutative expression is not easy to implement inside a runtime system because it leads to more complex dependencies, and the final DAG is generated on the fly during the execution. Figure 4.4 shows an execution DAG with dissociated operators and without the expression of the commutativity. We see that it creates extra dependencies compared to the previous graphs.



(a) Complete operators **(b)** Dissociate operators

Figure 4.3: FMM DAG with commutative. The commutative are expressed by the *OR* which means that several task are modifying the same data but that the order is not important.

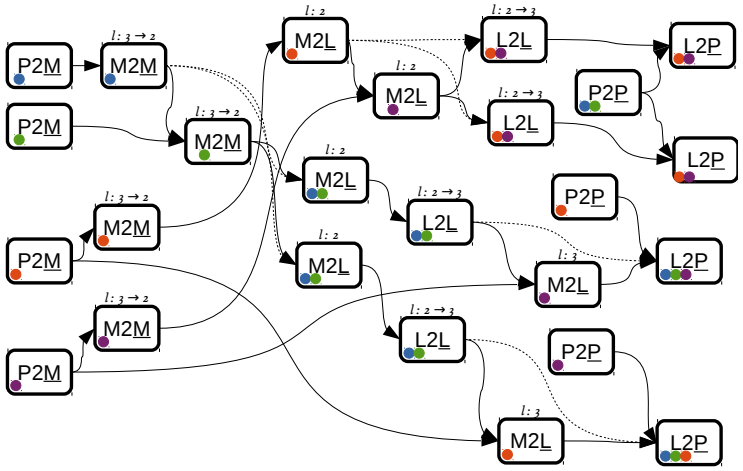


Figure 4.4: FMM DAG without commutative and with dissociate operators. Dashed edges represent meaningless information that should be removed using transitive reduction (TR).

4.4.2 Tasks-and-Dependencies FMM

We present in Section 2.2.1 the advantages of using a data-flow declaration to decompose a problem into tasks. In our presentation, we use OpenMP 4 *task depend* statements to express our data/tasks dependencies, but the approach is valid with any other runtime. However, at the time of writing, OpenMP does not support the commutativity expression and the FMM DAG is finally the one presented in Figure 4.5. Therefore, the order of insertion of the task is critical and directly impacts the DAG. Without the commutative propriety, it is important to insert the $P2P$ tasks before the $L2P$ otherwise the near-field is not be mixed with the far-field. A similar choice has to be made to choose whether the $M2L$ or the $L2L$ modify the local part of the cells first.

Starting from the task-based *fork-join* algorithm, we have to specify all the data access modes that each task has on its parameters. On the other hand, we remove the barriers because the runtime is in charge of the coherency. The resulting algorithm is presented in Algorithm 16 where a single thread inserts all the tasks. We make the choice not to dissociate the operators; this choice increases the granularity of the tasks but also increases the number of dependencies. As a result,

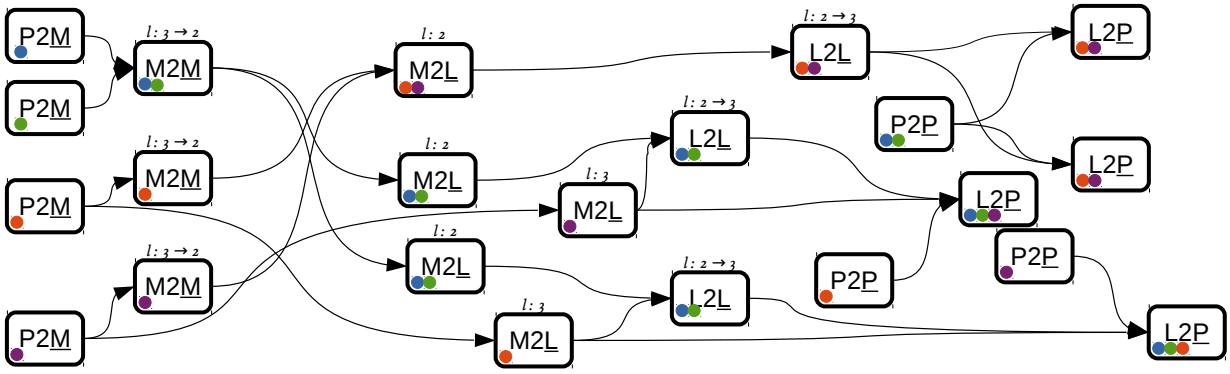


Figure 4.5: FMM DAG without commutative and not dissociate operator. We decide to insert the $M2L$ at level l before the $L2L$ from $l-1$ to l and to insert the $P2P$ before the $L2P$.

a $M2L$ has up to $189+1$ dependencies, a $M2M/L2L$ up to $8+1$ and a $P2P$ up to $26+1$. In our pseudo-code, we hid this large and variable number of dependencies but in the real implementation we need to use different pragma statements for each possibility. Finally, with OpenMP we cannot orient the execution using priorities.

Algorithm 16: FMM TaskDep Algorithm

```
function FMM(tree, kernel)
    #pragma omp parallel
        #pragma omp single
            // Near-field
            P2P_taskdep(tree, kernel);
            // Far-field
            P2M_taskdep(tree, kernel);
            for l = tree.height-2 → 2 do
                M2M_taskdep(tree, kernel, l);
            for l = 2 → tree.height-1 do
                M2L_taskdep(tree, kernel, l);
            for l = 2 → tree.height-2 do
                L2L_taskdep(tree, kernel, l);
            // Merge
            L2P_taskdep(tree, kernel);
            #pragma omp taskwaits
```

```
function M2M_taskdep(tree, kernel, level)
    foreach cell cl in tree.cells[level] do
        #pragma omp task depend(inout:cl.multipole) depend(in:tree.getChildren(cl.mindex, level).multipole)
        kernel.M2M(cl.multipole, tree.getChildren(cl.mindex, level));
```

```
function M2L_taskdep(tree, kernel, level)
    foreach cell cl in level.cells do
        #pragma omp task depend(inout:cl.local) depend(in:tree.getInteractions(cl.mindex, level).multipole)
        kernel.M2L(cl.local, tree.getInteractions(cl.mindex, level, Multipole));
```

FMM Critical Path

The bottleneck of the FMM comes from reduction of the workload at the top of the tree and the downward pass. For example, in the dense FMM the number of cells is divided by 8 as we go up in the tree, and depending on the kernels the work is also divided by the same coefficient. In consequence, the amount of parallelism reduces drastically at the top, and the upper levels must be moved downward by the $L2L$ to end the far-field. Therefore, while the $P2M$ tasks are relaxing potential $M2L$ (at the leaf level) and $M2M$ from $h-1$ to $h-2$, it is important to ensure that the $M2M$ is proceed first. For the same reason, the $M2L$ at the top are more critical than the ones at the leaf level. Of course, using tasks-and-dependencies, the $M2L$ at the leaf level are here to ensure that no thread remains idle, but it appears clearly that the execution path is important.

Priorities let orient the execution by deciding which tasks should be computed first among all the available/ready tasks. A possible set of priorities is given by the following order: $P2M$, $M2M$, $M2L$ above leaf level, $L2L$, large $P2P$, $M2L$ at the leaf level, $L2P$ and small $P2P$. If the underlying priority system supports numerous values, it is possible to give more precise priorities. For example, for the $M2L$ above the leaf level, we may use $h-4$ priorities to assign one priority per level (from level 2 to $h-2$).

Reduction Operation/Access Mode

In Section 2.2.2, we explain why the *commute* mode may constrict the parallelism. In our FMM algorithm, any *commute* access can be replaced by a *reduction* and this leads to new execution DAG. However, the overhead of the *commute* from the data replication might lead to poor performance. Therefore, the possible benefit of this mode depends on both the problem and the runtime system implementation.

4.4.3 Group-Tree Data Structure

The management of the dependencies makes the *tasks-and-dependencies* model more expensive than the *fork-join* approaches. The cost is decomposed in a static cost and a variable cost depending on the number of tasks and data dependencies.Therefore, it is interesting to increase the granularity of the tasks to reduce the static cost relatively to the task cost, and to reduce the variable cost by reducing the number of task and data dependencies. In a different context, it improves the granularity of the tasks and the number of data movements that are important issue with the accelerator devices.

We propose to use a group-tree data structure as shown in Figure 4.6. Starting from an usual octree, we aggregate n_g leaves or cells together and call such a block a *group*. It divides the number of data managed by the runtime by a factor n_g. We expect also to reduce the number of dependencies by the same factor, but the theoretical worst case has potentially even more dependencies: a group of n_g cells could need up to $189 \times n_g$ groups for the $M2L$. The scalar dependencies are then replaced by group dependencies: if an operation has a dependency on a cell cl, it is replaced by a dependency on the group that contains cl.

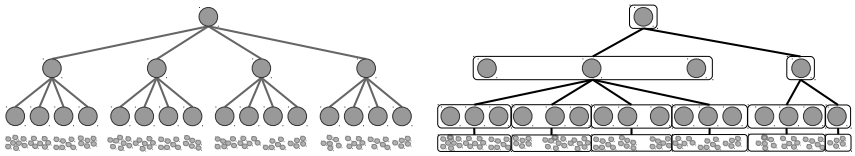


Figure 4.6: Group-tree with $n_g = 3$. The n_g contiguous existing cells/leaves belong to the same group.

Group-Tree Implementation

In this section we describe our implementation of a group-tree. We split the symbolic, multipole and local parts in different block of data. This allows to express the dependencies on the different parts of the cells/leaves. We store one triple per group with one linked list per level as in Figure 4.7.

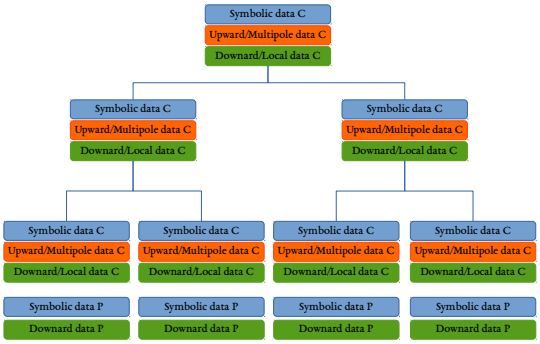


Figure 4.7: A group-tree where the symbolic, multipole and local data are allocated in separate memory blocks. We store the blocks with h linked lists (one per level) with 3 data pointers per list-node for the cells (C) and 2 data pointers per list-node for the particles (P).

The particle-group symbolic block and the cell-group symbolic block are similar. They are both composed of the number of leaves/cells, the Morton indexes of the first and last leaves/cells and the Morton indexes of all the existing leaves/cells included in the block. An example of symbolic block is shown in Figure 4.8 for a group of cells and in Figure 4.9 for a group of particles. For a group of cells, the multipole and local data blocks are usually an array of floating point values. If we have to store $\underline{m}$ and $\underline{l}$ values for the multipole and local parts of a cell, then the blocks are arrays of size $n_g \times \underline{m}$ and $n_g \times \underline{l}$, where n_g is the blocking parameter. Therefore, the multipole data of the i^{th} cell of a group is accessed by a direct indexing $multipole[i \times \underline{m}]$. The data block for the particles contains the successive vectors of the leaves that are included in the group. To access the potential values of the j^{th} particles of the i^{th} leaf of a group, we write $potential[offset[i] + j]$. In the case of a dense octree, the cells included in a group have consecutive Morton indexes and if the first cell has index f, then the block includes the Morton interval $[f; f + n_g - 1]$. However, in a sparse tree, the Morton interval covered by a group is much higher than n_g and gives $n_g << l - f$, with l the Morton index of the last cell of the block. We call the degree of filling of a group the ratio $n_g/(l - f + 1)$.

Complexities. The group-tree is designed to have an efficient linear access. Accessing the elements of a group consecutively is constant and so iterating on all the leaves/cells of a group has a

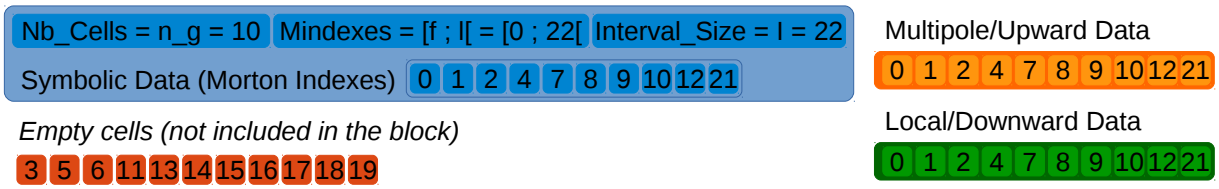


Figure 4.8: A Group of cells

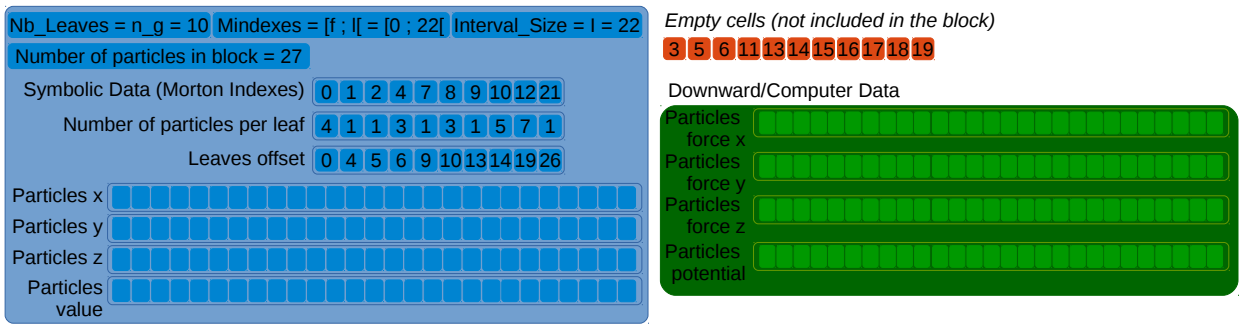


Figure 4.9: A Group of particles

complexity of $O(n_g)$. We store our group inside a linked list which lead to a linear complexity to iterate on the elements. Finding a cell is more expensive: we first seek the corresponding group with a cost of $O(N^l/n_g)$, where N^l is the number of cells at level l. Seeking a cell inside a block is done using binary search in $O(logn_g)$. This complexity could be reduced by using a search/binary tree above the groups but we almost never ask for a cell directly in our algorithm.

Group-Tree Construction

The group-tree is easy to construct with a bottom to top method. We start by computing the Morton indexes of the particles and we sort them in Morton increasing order. Then, we iterate and create a group of leaves for each n_g different Morton indexes. We proceed similarly, level by level, to create the parent blocks.

Sorting the particles has a $O(N^p logN^p)$ complexity with N^p the number of particles. Creating the groups at the leaf level has a linear complexity of $O(N^p)$ in time and $O(n_g)$ in space. At level l, generating the cell groups has a complexity proportional to the number of cells of the lower level $O(N^{l+1})$.

Group-Tree Extensions

We propose two extensions to our group-tree for special classes of problems. In some FMM applications, the cells or the leaves may have different sizes. In these cases, we use two extra arrays in the symbolic data block to save the sizes and the offsets of the elements.

In addition, it exists some FMM kernels (i. e. in acoustic or oscillatory kernels) which have operators that become more expensive as we go up in the tree. If we keep the same group size per level, the cost of the tasks decreases as we go up in the tree. We introduce a dynamic n_g to have one upper group by child group. Therefore, for instance, if at level $l+1$ we have two groups

that cover the Morton index intervals $[g_1^s, g_1^e]$ and $[g_2^s, g_2^e]$. We create a first parent group that covers $[parent(g_1^s), parent(g_1^e)]$ and if needed a second parent block that covers $[Max(parent(g_2^s), parent(g_1^e)+1), parent(g_2^e)]$.

4.4.4 Task-Based with a Group-Tree

In this section, we describe the FMM operators based on the group data structure.

Particles-cells operations: $P2M/L2P$. From our group-tree definition, the groups of particles at the leaf level and the groups of cells have the same elements. The groups of particles cover n_g leaves whereas the groups of cells cover n_g cells, and there is direct matching between this two kind of groups.To compute the $P2M$ and the $L2P$, we iterate on the groups of particles and the groups of cells at the leaf level. For each pair of groups, we then iterate linearly on the existing element in Morton index increasing order to apply the operator to the leaf and the corresponding cell. Therefore, it is a linear complexity in the linked list and inside the groups.

Level to level operations: $M2M/L2L$. Each group located above the leaf level has between 1 to 9 children groups if the group factor n_g is the same at all levels. Therefore, we iterate on the parent level and the children level at the same time. We increment the iterator to access all pairs of parent/children groups and we call the appropriate operator. Inside the operators, we iterate linearly on the parents and the corresponding children. The first element has to be found by dichotomy in $O(logn_g)$ but then the complexity is linear. To know the elements to compute we have to use the Morton index intervals. For a parent group covering the interval $[g_1^s, g_1^e]$ and a child group covering $[g_2^s, g_2^e]$, we apply the operator from parent with index $Max(g_1^s, parent(g_2^s))$ to parent of index $Min(g_2^e, parent(g_2^e))$.

Transfer operations: $M2L/P2P$. The transfer operations involve two kinds of computations: the inner operation computes the interactions inside a group and the outer operation computes the interactions between groups. For the inner computation, we compute the interaction list for each element on the fly and we find the Morton indexes that are included in the group. We remind that finding a cell in a group has a complexity of $log(n_g)$, so the total complexity is $O(n_g \times 189 \times logn_g)$ for the $M2L$ and $O(n_g \times 26 \times logn_g)$ for the $P2P$. For the outer computation, we need to store the interactions that each group has with the others, we call the list of interactions between two groups the interactions table. This table tells where the elements are located in the groups, it gives their Morton indexes and the relative position of each interaction which is a value between 0 and 342 for the $M2L$ ($[-3;+3] \times D$) or 0 and 26 for the $P2P$ ($[-1;+1] \times D$). The outer computation between two groups is done by a single thread in a task and thus there is no race condition or critical section. Figure 4.10 shows the different operators and their data access mode.

Building the interaction tables. To construct the interaction tables, we first iterate through the elements of a group and compute their interaction lists. Then, we sort all the interactions of the

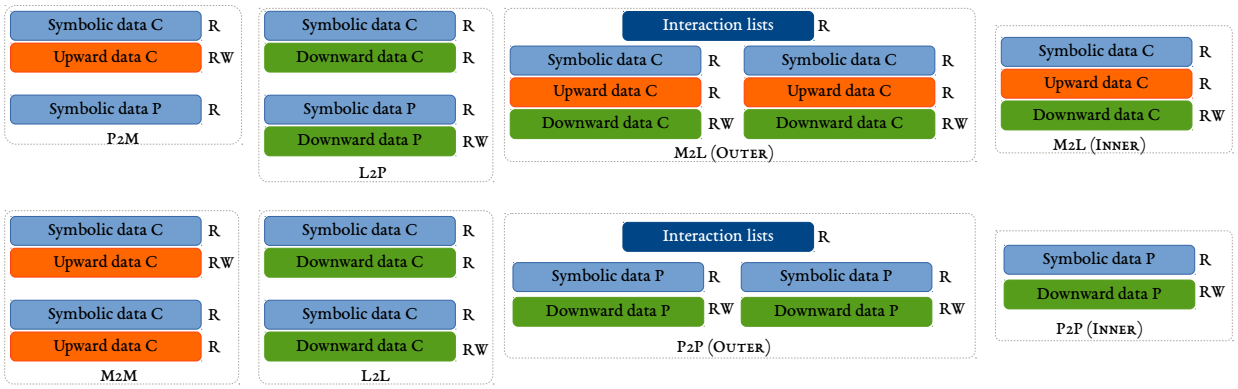


Figure 4.10: Data accesses for the Group data structure. R stands for read and RW for read-write.

group based on the Morton indexes of the external elements. Finally, we divide the array to match the different existing groups. These operations are easily parallelized using the *fork-join* models, but it would be nice to parallelized them using tasks and dependencies and to insert them in the DAG.

4.5 Distributed Tasks-and-Dependencies

The first step consists in distributed the data among all the processes as in the Hybrid OpenMP/MPI implementation presented in Section 4.3. Each process hosts the particles inside a Morton index interval and create its local group-tree. The processes exchange the information of their groups of elements; each process has a data structure which tells what groups exist on the other processes and what Morton intervals they cover but not if a cell exists inside a remote group. The DAG of the distributed FMM is shown in Figure 4.11 where the dependencies between the processes have been replaced by sends/receives. A send is a task that performs a read on the data and a receive is a task that performs a write.

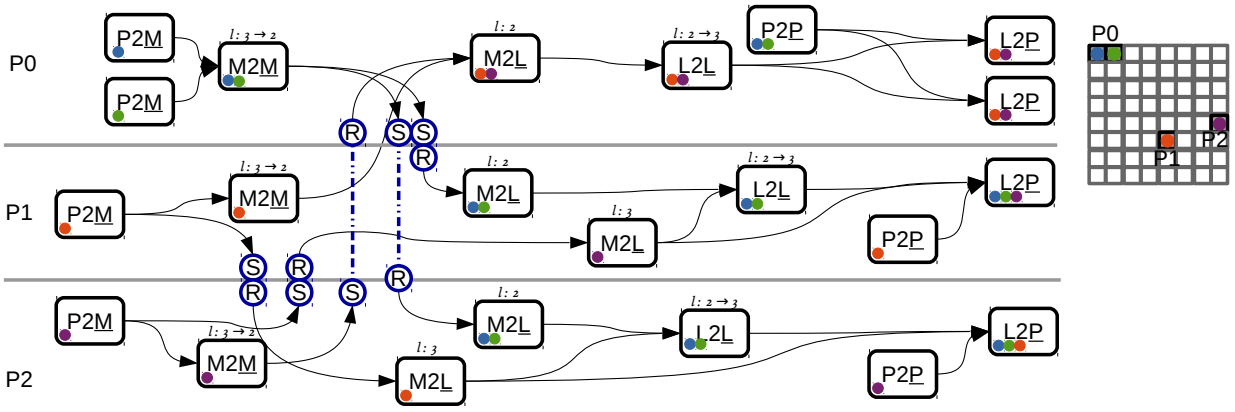


Figure 4.11: FMM DAG in $2D$ with a tree height $h = 4$. The communications are shown by R (read/send) and W (write/receive).

In our implementation we use StarPU-MPI introduced in Section 2.2.3. During a preprocessing stage, we find which groups should be sent to or received from which processes and we post all

the receives with a StarPU-MPI receive-detached function. Then during the FMM computation, we post the sends using a StarPU-MPI send detached function.

4.6 Enabling Accelerators in Runtime-Based FMM

To incorporate accelerators in an application that is based on StarPU we define one function pointer per device/processing unit (results are not shown in the current study). StarPU detects and initializes the device according to the user requests and manages the data movements between the host and the device ensuring the data coherency. In terms of implementation, we must use plain old data (POD) to allow StarPU move/duplicate the memory blocks (POD are raw data without pointers). In addition, we have to create the operator kernels for the different target architectures.

4.7 Particle Simulation Parallel Study

To illustrate the behavior of the different parallelization strategies, we present parallel efficiency and speedup results in shared and distributed memory. We compute interactions between particles for the $1/r$ kernel and the far-field is approximated by a Legendre expansion with spherical harmonics. This kernel is accelerated by rotation as described in [100; 101]. We consider an expansion of order 5 which gives an accuracy around 10^{-3}. We test two different distributions; an uniform distribution inside a cube (*uniform*) and a non uniform distribution on the surface of an ellipse ($non - uniform$) illustrated by Figure 4.12a and Figure 4.12b respectively. The nodes are composed by 2 Dodeca-core Haswell Intel® Xeon® E5-2680 at $2,50GHz$ and $128GB$ (DDR4) of shared memory. We use Gcc 4.9.2 and Openmpi 1.8.4.

(a) Cube (*uniform*) **(b)** Ellipsoid ($non - uniform$)

Figure 4.12: Tests cases Particles distributions.

4.7.1 Shared Memory Parallelization

We compare the strategies for different number of particles for the uniform distribution in Figure 4.13 and the non-uniform distribution in Figure 4.14. The strategies are described in the following paragraphs.

Classical approaches

- *parallel-for.* This strategy is presented in Section 4.2.1 where each operator is divided among the threads with a *parallel-for*. It is based on the Gcc OpenMP implementation with a chunk size of 10 and a dynamic schedule.
- *Section tasks-and-wait.* In this strategy, we mix the near and far fields with a two task-sources pattern as presented in Section 4.2.3. It is based on the Gcc OpenMP implementation and uses a chunk size of 1.
- *parallel-for Balanced.* Here we extend the approach from *parallel-for* presented in Section 4.2.1 with a different division of the work. We divide the loops between the threads by balancing the workload. We use a loop static schedule and using a greedy algorithm to allow each thread to compute about the same number of interactions. It is based on the Gcc OpenMP implementation.
- *tasks-and-dependencies.* This is a single granularity expression of the FMM as described in Section 4.4.2 (there is no group-tree). It is executed over GCC OpenMP 4 implementation and therefore there is no support of the commutative or reduction modes. The $M2L$ at a level l are inserted before the $L2L$ from $l-1$ to l and the $P2P$ are inserted before the $L2P$.

Tasks-and-dependencies approaches

- *GT OpenMP-4.* This implementation uses a group-tree with a *tasks-and-dependencies* as presented in Section4.4.4 over the GCC OpenMP 4 implementation. That is why, there is no commutative or reduction operations in this implementation.
- *GT StarPU (NC-NR).* This implementation is the same as the GT OpenMP-4 but using StarPU and priorities. Similarly, there is no commutative (NC) and no reduction (NR) operation.
- *GT StarPU (NR).* This strategy extends the GT StarPU (NC-NR), and uses the commutative operations but not the reduction (NR).
- *GT StarPU (commutative L2L).* In this configuration, we use the commutative and the reduction operations. The commutativity is used everywhere even for the $L2L$ from $h-2$ to $h-1$.
- *GT StarPU.* In this configuration, we use the commutative and the reduction operation, but we replace the data mode access of the $L2L$ from $h-2$ to $h-1$ by a read-write access instead of a commutative access. It means that the $M2L$ at the leaf level will start once the work in the top of tree is over (locally).

Classic parallelizations For both distributions, see Figure 4.13 and Figure 4.14, the *tasks-and-dependencies* implementation is always the slower one. Even in sequential, this implementation is slower whereas the other are roughly similar, and the difference increases with the number of threads which gives us an important information; the current OpenMP-4 GCC implementation of the tasks-and-dependencies module is to expensive in terms of dependencies/tasks management. It is true that this implementation is difficult for the runtime system because there are up to $189+1$ dependencies per task and the granularity is reduced to a single element. Moreover, there is not

commutative expression with OpenMP and the resulting parallelism is highly constrained.

The *parallel-for Balanced* implementation is competitive in all uniform cases, and the fact that we have a static division of the work looks to be advantageous in the small test cases where the synchronization between threads should be avoid. On the other hand, for the larger non-uniform distribution from Figure 4.14h, the static distribution of the work is not optimal and it makes the *parallel-for Balanced* slower than the *parallel-for*. The division of the work is done on contiguous intervals but the high density areas are concentrated at the poles of the ellipse. Therefore, it is more appropriate to split this area in small pieces as we do in the dynamic schedule in the *parallel-for*.

The objective of the *tasks-and-wait* strategy is to remove potential barriers by having two sources for the task insertions. Again the fine granularity leads to low performances. Moreover, the absence of priority system in OpenMP transforms this pattern in a basic task and wait execution.

The basic *parallel-for* implementation shows good results even for non-uniform test cases. This is mainly because the underlying tasks have good granularities and the division of the work is local; if there are two regions in a simulation, one with a high density the other with a low density, this approach will not divide the entire interval among the threads but rather uses very small divisions.

Group-Tree based parallelizations The GT OpenMP-4 executions are different from the GT StarPU (NC-NR) even so they use the same data structure and do not use the reduction or commutative operations. More precisely, GT OpenMP-4 appears to have inconsistent results and does not scale in most of the cases whereas the GT StarPU (NC-NR) shows good performances. Therefore, the difference come mainly from the underlying runtime systems and their respective overhead. In addition, in StarPU all the tasks are inserted and managed together, whereas in GCC OpenMP their is an insertion window which may hide some potential ready tasks. Surprisingly the extensions of the GT StarPU (NC-NR) which include the commute, GT StarPU (NR), or the commute and the reduction, GT StarPU, do not give better results. This means that their is enough parallelism in the given test cases. However, the GT StarPU (commutative L2L) strategy is less efficient which means that using the commute on the last level $L2L$ degrades the performance. The reason is related to the StarPU system to manage the commutative tasks which is done before the priority are taken into account. Therefore, when the mode between the last level $L2L$ ($h-2 \rightarrow h-1$) and the last level $M2L$ ($h-1$) are commutative, the $M2L$ tasks are ready just after the $P2M$ and are selected by StarPU. This automatically makes the last level $L2L$ less prioritized even so they are more critical.

By comparing these different strategies, it appears that there is not a perfect scheme but the *parallel-for* and GT StarPU have a good efficiency. The main advantage of the *parallel-for* is its portability and simplicity whereas the GT StarPU can easily exploit the accelerators. But the other approach are tied to the current GCC OpenMP implementation and its overhead in the tasks management or the lack of a priority system.

In Figure 4.15, we show the execution trace for GT StarPU and 24 threads. We see that despite the non-uniform distribution, the execution is correctly pipelined and there is no idle time.

4.7.2 Distributed Memory Parallelization

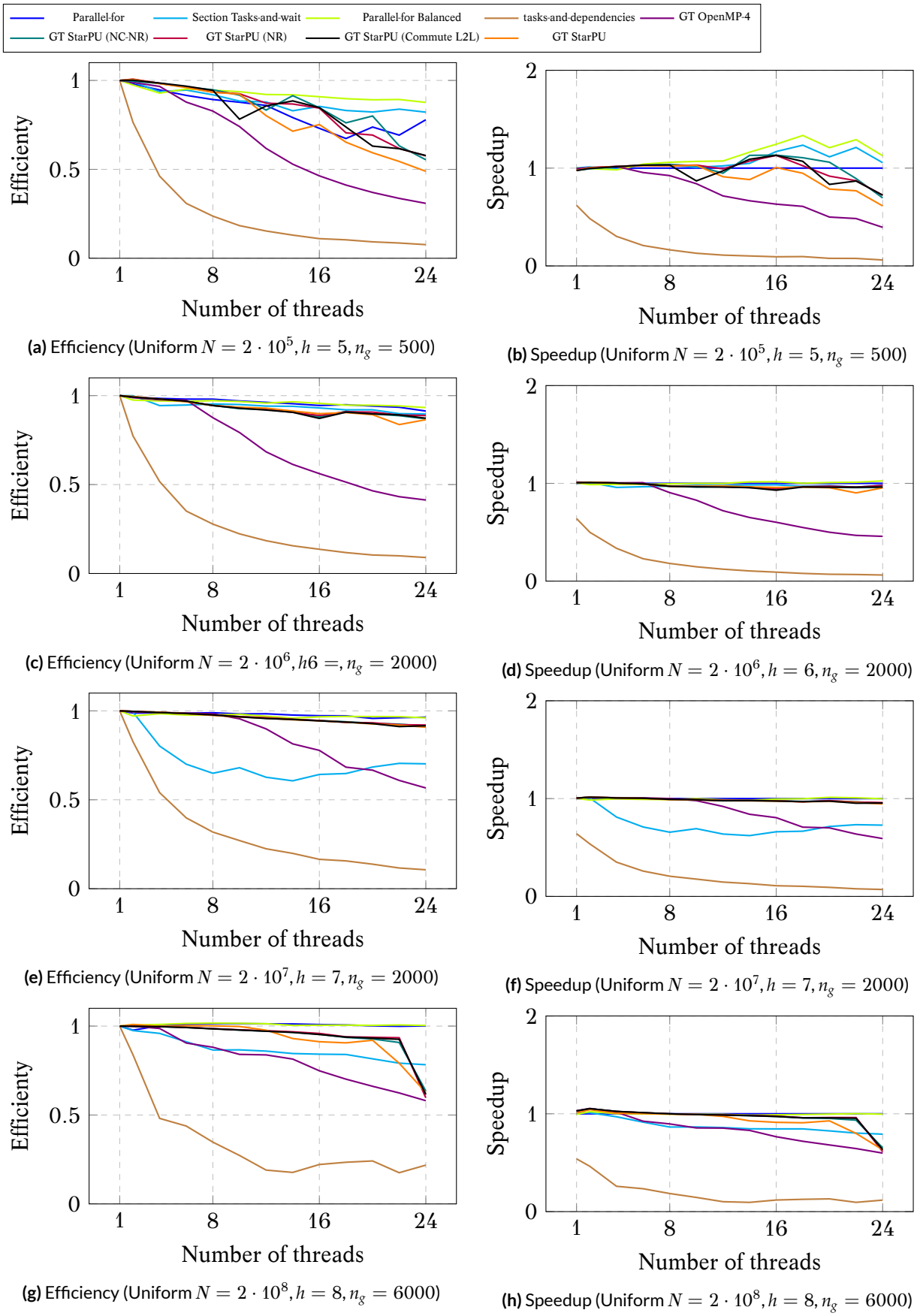
In this section, we compare the Hybrid MPI/OpenMP implementation presented in Section 4.3, which is described as a fork-join strategy with communication hiding, against a pure task-based approach using StarPU (StarPU-MPI) as described in Section 4.5 for the uniform test case.

From Figure 4.16, both implementations have very close results and good efficiencies. For the smaller test cases, see Figures 4.16a and 4.16b, the StarPU-MPI implementation is slower and its efficiency is decreasing as the number of nodes increases. In fact, the test case is small (less than a second in parallel) and thus the overhead from the runtime system becomes nonegligible. Moreover, the granularity of the tasks should be reduced to have enough parallelism but this makes the overhead even more important. For the larger test case, both implementation are very close which means that the computation and communication are correctly interleaved.

In Figure 4.17, we show an execution trace for the StarPU-MPI using 7 nodes. We see that the work is correctly pipelined even though there is a small difference at the end between the different processes/threads. The receives are posted at the beginning of the executions, and thanks to StarPU once a data is computed it is sent without impacting the execution.

4.8 Summary

We present different parallelization strategies for the FMM in shared and distributed memory. The advanced methods using runtime systems need a larger granularity and few dependencies to be efficient. Therefore, the usage of the group-tree helps to amortize the runtime overhead. Using basic access modes as read/write limits the parallelism and we also need advanced access mode as commutative or reduction operations. While, the classic approaches, *parallel-for* and hybrid MPI/OpenMP, are very competitive, the StarPU strategies are easily expendable to accelerators.



(a) Efficiency (Uniform $N = 2 \cdot 10^5, h = 5, n_g = 500$)

(b) Speedup (Uniform $N = 2 \cdot 10^5, h = 5, n_g = 500$)

(c) Efficiency (Uniform $N = 2 \cdot 10^6, h6 =, n_g = 2000$)

(d) Speedup (Uniform $N = 2 \cdot 10^6, h = 6, n_g = 2000$)

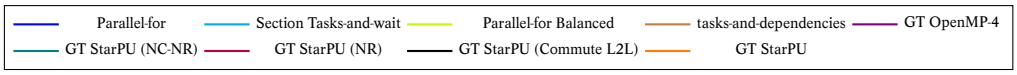
(e) Efficiency (Uniform $N = 2 \cdot 10^7, h = 7, n_g = 2000$)

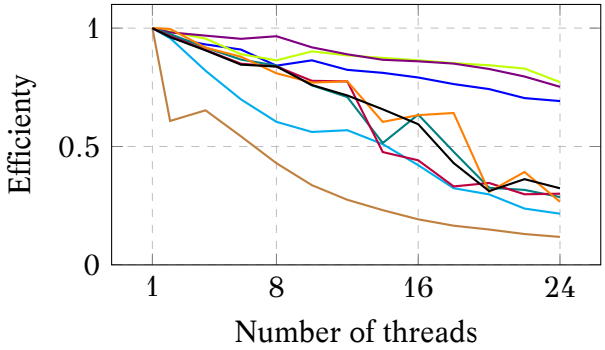
(f) Speedup (Uniform $N = 2 \cdot 10^7, h = 7, n_g = 2000$)

(g) Efficiency (Uniform $N = 2 \cdot 10^8, h = 8, n_g = 6000$)

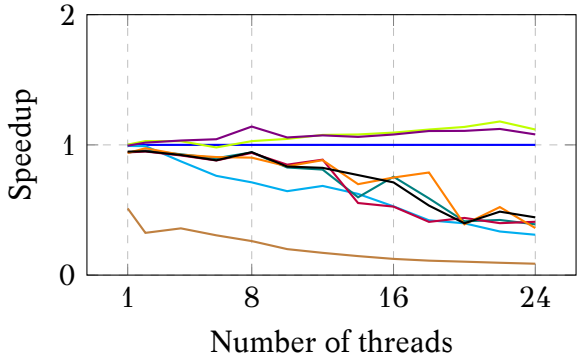
(h) Speedup (Uniform $N = 2 \cdot 10^8, h = 8, n_g = 6000$)

Figure 4.13: ScalFMM shared memory results for the uniform distribution from 1 to 24 threads and for 4 different number of particles. The execution times for the *parallel − for* strategy using 24 threads for the respective cases are $0.07s$,$0.7s$, $7.7s$ and $88s$.

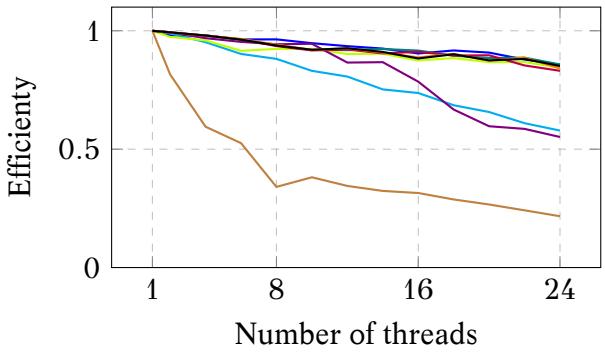




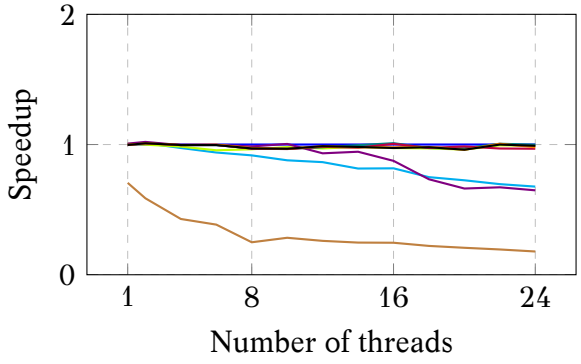
(a) Efficiency (Ellipsoid $N = 2 \cdot 10^5, h = 8, n_g = 200$)



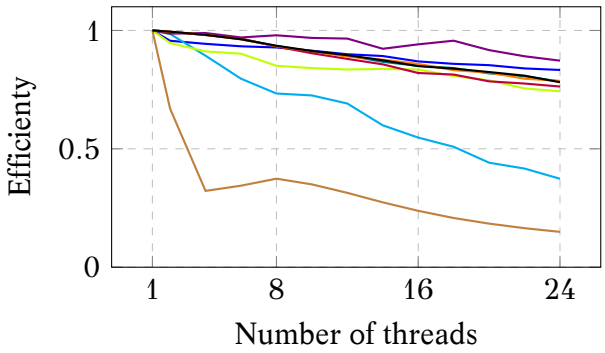
(b) Speedup (Ellipsoid $N = 2 \cdot 10^5, h = 8, n_g = 200$)



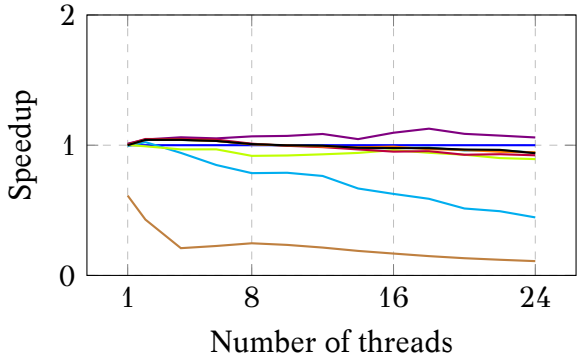
(c) Efficiency (Ellipsoid $N = 2 \cdot 10^6, h = 9, n_g = 1400$)



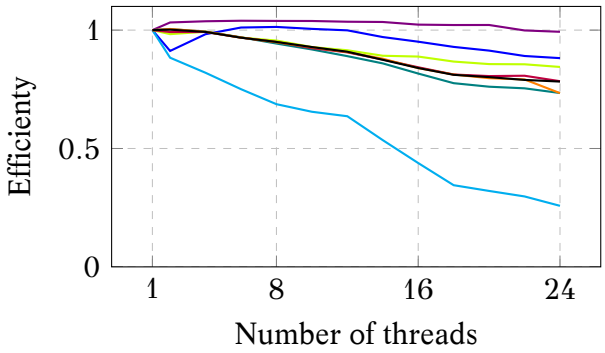
(d) Speedup (Ellipsoid $N = 2 \cdot 10^6, h = 9, n_g = 1400$)



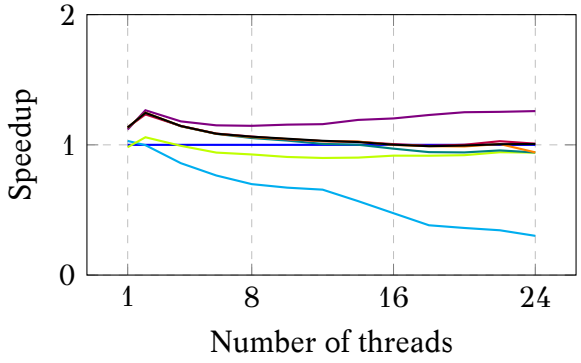
(e) Efficiency (Ellipsoid $N = 2 \cdot 10^7, h = 11, n_g = 2000$)



(f) Speedup (Ellipsoid $N = 2 \cdot 10^7, h = 11, n_g = 2000$)



(g) Efficiency (Ellipsoid $N = 2 \cdot 10^8, h = 13, n_g = 8000$)



(h) Speedup (Ellipsoid $N = 2 \cdot 10^9, h = 13, n_g = 8000$)

Figure 4.14: ScalFMM shared memory results for the ellipsoid non-uniform distribution from 1 to 24 threads and for 4 different number of particles. The execution times for the *parallel — for* strategy using 24 threads for the respective cases are $0.05s$,$0.49s$, $4.8s$ and $58s$. Remark: the tasks-and-dependencies strategies cannot execute the $N = 2 \cdot 10^9$ test case because the GCC OpenMP 4 implementation is not able to manage the large amount of tasks and their dependencies.

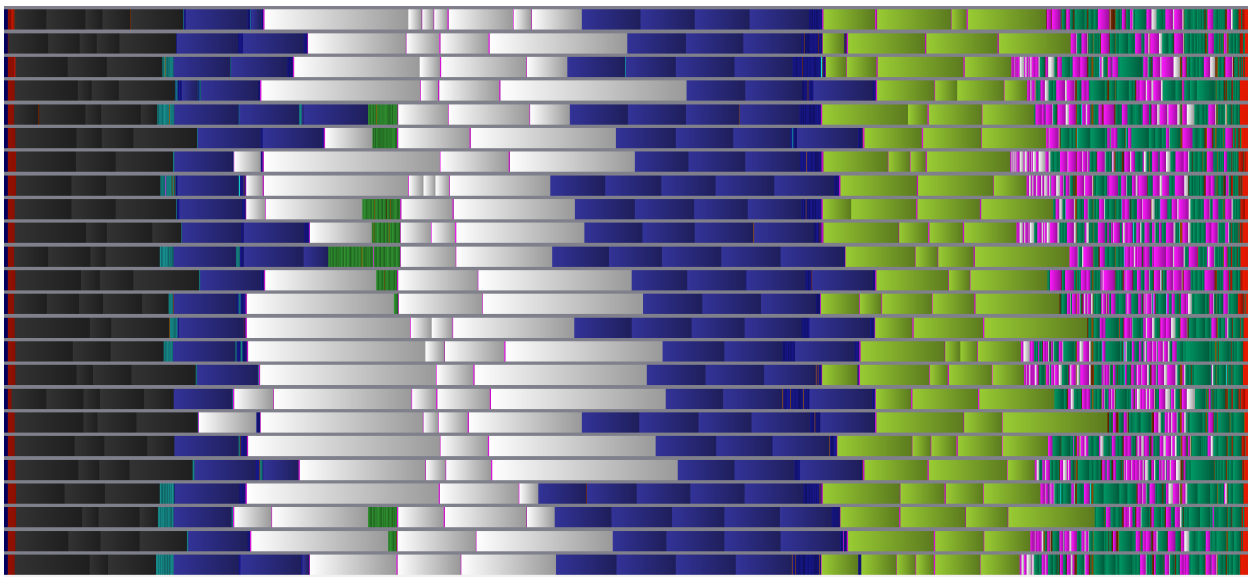
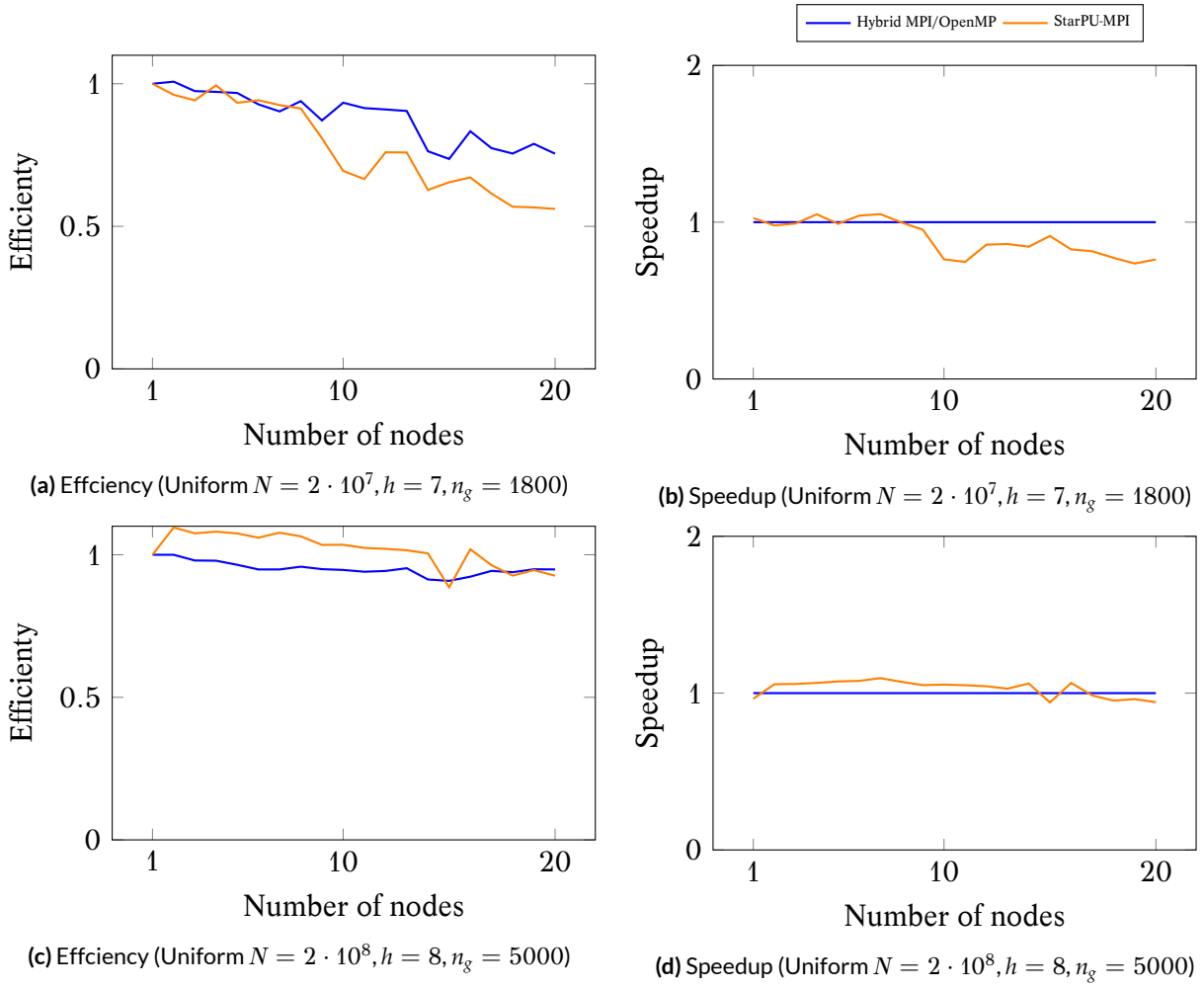


Figure 4.15: Shared memory execution trace with the configuration: $N = 2 \cdot 10^7$ particles, non uniform ellipsoid distribution, height of the tree $h = 11, 24$ threads and block size $n_g = 8000$ in $5.2s$. Legend: P2P ( ), P2P between groups ( ), P2M ( ), M2M ( ), M2L ( ), M2L between groups ( ), L2L ( ) and L2P ( ).



(a) Effciency (Uniform $N = 2 \cdot 10^7, h = 7, n_g = 1800$)

(b) Speedup (Uniform $N = 2 \cdot 10^7, h = 7, n_g = 1800$)

(c) Effciency (Uniform $N = 2 \cdot 10^8, h = 8, n_g = 5000$)

(d) Speedup (Uniform $N = 2 \cdot 10^8, h = 8, n_g = 5000$)

Figure 4.16: ScalFMM distributed performance study for the uniform distribution using 24 threads and 1 to 20 nodes. The execution times for the Hybrid MPI/OpenMP strategy using 20 nodes for the respective cases are $0.57s$ and $5.1s$.

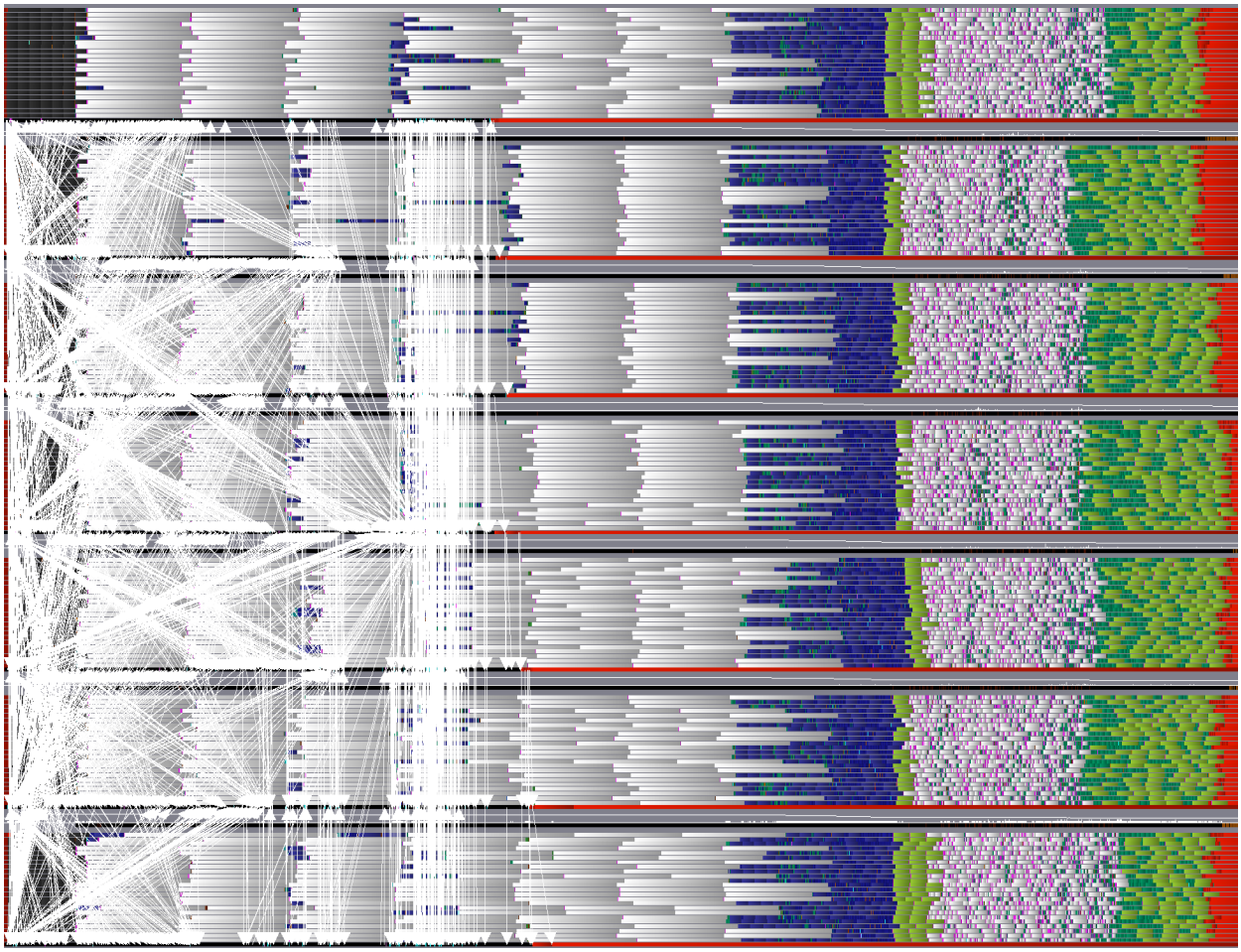


Figure 4.17: Distributed memory execution trace for the StarPU-MPI with the configuration: $N = 2 \cdot 10^8$ particles, uniform distribution, height of the tree $h = 8$, 24 threads, 7 nodes and block size $n_g = 2000$ in $13.5s$. The arrows represent MPI communications. The 24 threads of nodes are represented contiguously, whereas the nodes are separated by gray lines. Legend: P2P ( ), P2P between groups ( ) , P2M ( ) , M2M ( ) , M2L ( ), M2L between groups ( ), L2L ( ) and L2P ( ) .

5

Time-Domain BEM Solver using the FMM

In this chapter, we study the acceleration of the time-domain BEM for the wave equation with the Fast Multipole Method. In the first part, we present the algorithm details of our TD-FMM kernel which has been taken from [3; 102]. Then, we provide the formulation of the different FMM operators. Finally, we discuss the time/frequency expression of the kernel and give some results using ScalFMM as parallelization engine.

5.1 Time-domain BEM FMM Formulation

5.1.1 Limits of the Direct Approach

In Section 1.2.1, we describe the direct resolution algorithm and the linear system of our time-domain BEM. This matrix solve has a quadratic complexity at each step: the K^{max} matrices are sparse but lead to $d \times N^2$ NNZ values with $d \geq 1$. The complexity for the complete computation is $O(T.N^2)$ with T the number of time steps. However, this description does not include the construction of the interaction matrices which has a quadratic complexity too. To build these matrices, we have to compute the relations in time and space that each unknown has with each other. These two quadratic complexities may limit the size of the problems we are able to solve and we aim to reduce the whole complexity (the construction of the matrices and the solve) using the FMM.

5.1.2 Receiving from the Past or Propagating to the Future

The linear system expression of our TD-BEM has been presented in Section 1.2.1 and we remind here that at each time step we solve

$$\begin{aligned} \sum_{k\geq 0}^{K^{max}} M^k \cdot a^{n-k} &= l^n \,, \\ s^n &= \sum_{k=1}^{K^{max}} M^k \cdot a^{n-k} \,, \\ a^n &= (M^0)^{-1} \left(l^n - s^n\right) \,. \end{aligned} \tag{5.1}$$

In this approach, we compute what the unknowns receive from the previous time steps into the current summation vector s^n. In terms of accesses, at time n we use the $K^{max} - 1$ vectors a^{n-p}, with $1 \leq p \leq K^{max}$ and $n - p > 0$, which are the past emissions that impact the mesh in the present.

The same computation can be done with a different philosophy if instead of computing what the unknowns receive, we propagate the present result to the future states. At each time step, once we compute the current state a^n, we compute its propagation to the K^{max} next summation vectors, which is expressed by

$$s^{n+p} = s^{n+p} + (M^p \cdot a^n); \; \forall p, \; 0 < p \leq Min(K^{max}, T - n) \,. \tag{5.2}$$

The summation vectors are built incrementally by the previous $K^{max} - 1$ iterations relatively to their respective time. The vector a^n is used to update the $K^{max} - 1$ next summation vectors. Once it has been propagated, the vector a^n is not used any more and can be removed from the main memory. At time step n, the summation vector s^n is already complete, and we just subtract it to l^n and solve the result with M^0,

$$a^n = (M^0)^{-1} \left(l^n - s^n\right) \,. \tag{5.3}$$

This approach is presented in Figure 5.1

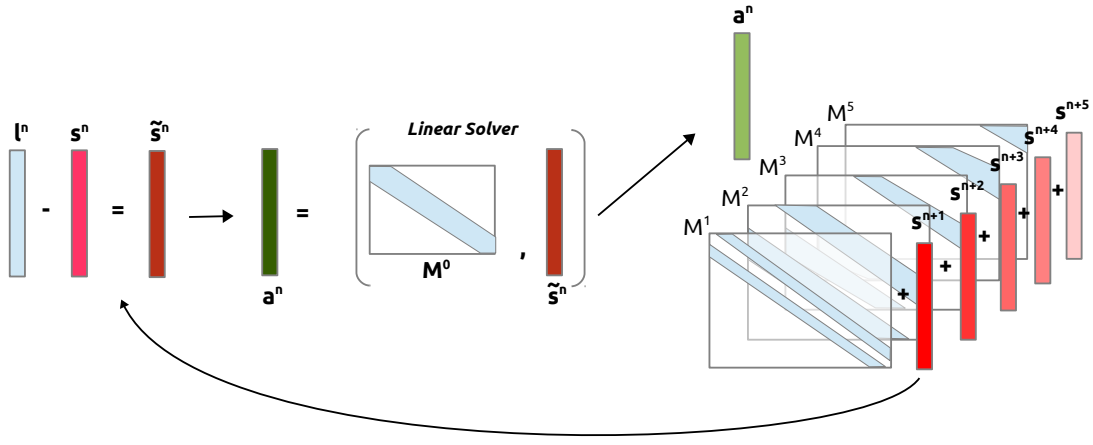


Figure 5.1: Propagation of the current state to the future

In our FMM kernel, we consider the same idea and spread the current state to the future. Figure 5.2 gives a simplified view of the objective where the emission to distant unknowns in time/distance is done by the FMM. The rest of the algorithm is similar to the matrix approach where we take into account the incident wave l and solve the linear system with M^0. However, the well-separated criterion between the unknowns is based on the spatial decomposition from the FMM octree.

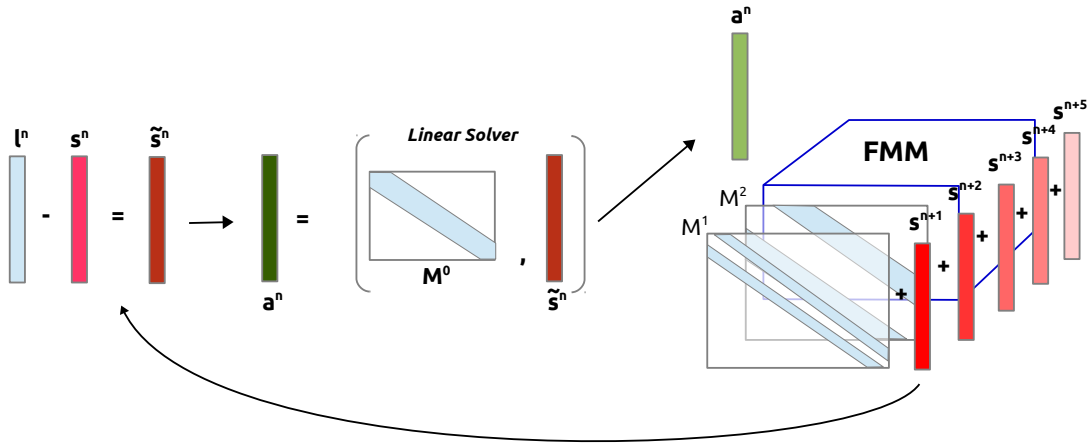
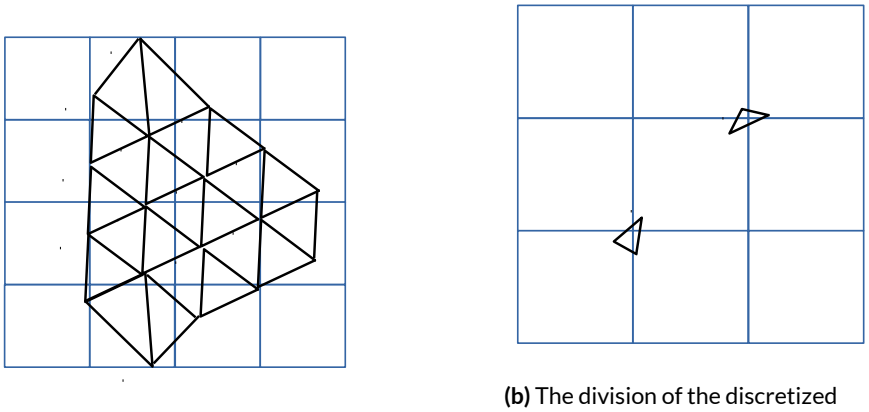


Figure 5.2: The time-domain BEM FMM in a schematic view

5.1.3 Spatial Division of the Mesh by the FMM Octree

Our mesh that is composed of surface elements and integration points has to be integrated into the FMM octree and its corresponding simulation box. The simulation box is the smallest cube which includes our mesh and which is parallel to the axes. The FMM octree is built over this region and creates a grid at the leaf level on the mesh as illustrated by Figure 5.3a. The surface discretization elements might be covered by more than one leaf, and thus we have to decide how to connect our mesh and the octree subdivision. The three most natural distribution are: 1) to distribute the elements based on their centers, 2) to distribute the edges of the elements or 3) to distribute the integration points (Gauss points). The choice is important because the accuracy decreases if close elements are computed by the FMM instead of the direct method. Figure 5.3b shows two triangles that both have a part inside the same leaf but also parts in distant leaves. Depending on the distribution, these two triangles are computed by a direct computation ($P2P$) or by the FMM ($M2L$ at the leaf level).

Based on the study [103], we distribute the elements using their integration points. The interactions between elements in neighbor leaves are computed using the matrix approach as presented in Chapter 3 but with a propagation to the future instead of a reception from the past. On the other hand, the interactions between elements distant by more than one leaf are approximated and computed using the FMM. The size of the cells/leaves and the spacial subdivision is based on the height of the tree h. The width of the leaves cannot be too small otherwise very close elements are computed by the FMM. To ensure a correct accuracy, we choose to have the leaves 5 times



(a) The octree leaf grid over a mesh (for $h = 3$)

(b) The division of the discretized mesh elements between the leaves. A choice has to be made on which leaf an element belongs to.

Figure 5.3: A mesh in an octree.

greater than Δx which is the size of the elements from the discretization of the mesh (or the average of the sizes). At the beginning of our application, we search the appropriate tree height h such that $5\Delta x \leq w_{h-1} = B/2^{h-1}$, where B is the width of the simulation box that cover the entire mesh and w_l the width of the boxes/cells at level l. From these definitions and the wave properties, c the velocity and λ the wavelength, we can define additional variables

- d or $d^l = \sqrt{3}w_l$: the diagonal of the box at level l.
- R or $R_l = d^l/2 = \sqrt{3}w_l/2$: the half of diagonal of the box at level l. It is also the radius of the sphere that includes this box.
- r_t or $r_t^l = int(R_l/(c\Delta t))$: the radius of the box in terms of $c\Delta t$ at level l.

5.1.4 Principle and Formulation

In this section, we describe the FMM kernel which is based on the formulation given in Appendix A. This section introduces the mathematical details of our problem formulation, but it is not a prerequisite to the understanding of our contribution and the implementation parts. Readers who do not feel concerned about how the propagation of the wave is approximated mathematically may concentrate on the sphere discretization (Section 5.1.5), the APS function description (Section 5.1.6) and the resulting FMM operators (Section 5.2).

Simplification of the Matrix Expression

The aim of the FMM approach is to accelerate the propagation of the present to the future, see Equation 5.2. We consider a function $\Phi(x,t)$ defined on the data of a^n, from Equation

$$\begin{cases} \Phi(x,t) = \sum_{1\leq j\leq N_S} \sum_{m\geq 1} a_j^m \gamma_m(t)\varphi_j(x), \\ P(x,t) = \sum_{1\leq j\leq N_T} \sum_{m\geq 1} b_j^m \gamma_m(t)p_j(x). \end{cases} \tag{5.4}$$

In this equation, a^n is the solution of the time step n, and is represented by the surface function Φ over the mesh.

From Φ and the test-function $\partial\Psi/\partial(x,t) = \varphi_i(x)\chi_m(t)$ later in time ($m > n$) we compute

$$\begin{aligned} s_i^m &= -\frac{1}{c}\int_{t\in\mathbb{R}}\int_{\Gamma\times\Gamma}\frac{\vec{n}(x).\vec{n}(y)}{4\pi|x-y|}\frac{\partial^2\Phi}{\partial t^2}(y,t-\frac{|x-y|}{c})\varphi_i(x)\chi_m(t)dxdydt \\ &= -\frac{1}{c}\int_{t\in\mathbb{R}}\int_{\Gamma\times\Gamma}\frac{1}{4\pi|x-y|}\vec{rot}_\Gamma\Phi(y,t-\frac{|x-y|}{c})\vec{rot}_\Gamma\varphi_i(x)\chi_m(t)dxdydt\,. \end{aligned} \tag{5.5}$$

It is appropriate to use the same approach as the frequency FMM which allows to compute fast matrix-vector product. From a vector $(t_i)_{1\leq i\leq n}$ to represent the function $\vec{t}(x) = \sum_{1\leq i\leq n} t_i.\vec{\varphi_i}(x)$, we compute the j^{th} component of the product $a.t$ by

$$(a.t)_j = -\frac{1}{ik}\oint_{\Gamma\times\Gamma}\frac{\partial^2 G}{\partial v_x\partial v_y}\Phi(y)\varphi_i(x)\,. \tag{5.6}$$

The core of the frequency matrix-vector product expression is replaced by the one from Equation 5.5, and in both cases there is a radiation on a surface element, with a pressure jump $\Phi(x,t)$ and flow $\vec{t}(x)$, to study its impact on the mesh Γ. The Equation 5.5 can be simplified because it is equal to $cD\Phi$ with D given by

$$s_i^m = D\Phi(x,t) = \frac{1}{4\pi}\oint_{\Gamma\times\Gamma}\frac{\partial}{\partial n_x\partial n_y}\frac{\Phi(y,t-|x-y|/c)}{|x-y|}dy\,. \tag{5.7}$$

In practice $cD\Phi$ is computed using Equation 5.5 to remove the singularities, but this is not a problem in our time-domain approach since we compute only far interactions. Therefore, the resulting equation to compute is given by

$$s_i^m = c\int_{t\in\mathbb{R}}\oint_{\Gamma\times\Gamma}\frac{\partial}{\partial n_x\partial n_y}\frac{\Phi(y,t-|x-y|/c)}{4\pi|x-y|}\varphi_i(x)\chi_m(t)dxdydt\,. \tag{5.8}$$

We consider G the Green kernel given by

$$G(R,t) = \frac{\delta(t-R/c)}{4\pi R}\,. \tag{5.9}$$

This kernel is the Fourier transform of the Green kernel of the frequency domain problem ($e^{ikR}/4\pi R$). Equation 5.8 can be written using a convolution in time, written $*$ for the rest of the study, as

$$s_i^m = c\int_{t\in\mathbb{R}}\oint_{\Gamma\times\Gamma}\frac{\partial}{\partial n_x\partial n_y}G(|x-y|,t)*\Phi(y,t)\varphi_i(x)\chi_m(t)dxdydt\,. \tag{5.10}$$

In order to simplify, we remove the normal derivatives, because it is possible to introduce them again later in the calculus, and we keep the same notation. Finally, we aim to compute

$$s_i^m = c\int_{t\in\mathbb{R}}\int_{\Gamma\times\Gamma}G(|x-y|,t)*\Phi(y,t)\varphi_i(x)\chi_m(t)dxdydt\,. \tag{5.11}$$

Kernel Decomposition

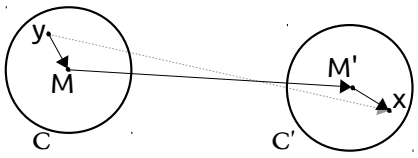


Figure 5.4: Distant interaction from x to y

Like in the frequency case, the Green kernel is decomposed with a configuration similar to the one shown in Figure 5.4. We denote by S the unit sphere in $\mathbb{R}^3$, $\vec{s}$ a point over S and $\vec{R} = \vec{xy}$ the vector between x and y. Let introduce the operator $\tilde{G}$ given by

$$\tilde{G}(R,t) = -\frac{1}{8\pi^2 c}\frac{\partial}{\partial_t}\int_{\vec{s}\in S}\delta(t-\frac{\vec{s}.\vec{R}}{c})d\vec{s}. \tag{5.12}$$

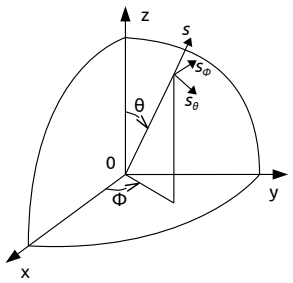


Figure 5.5: Spherical coordinate over the unit sphere S

On the unit sphere S a coordinate system is chosen such that the axis (Oz) is in the same direction as $\vec{R}$ as shown in Figure 5.5. We obtain the relation $\vec{s}.\vec{R} = Rcos\theta/c = \tau$ and $\tilde{G}$ writes

$$\begin{aligned}
\tilde{G}(R,t) &= -\frac{1}{8\pi^2 c}\frac{\partial}{\partial_t}\int_{\theta=0}^{\pi}\int_{\varphi=0}^{2\pi}\delta(t-\frac{Rcos\theta}{c})sin\theta d\theta d\varphi \\
&= -\frac{1}{4\pi^2 c}\frac{\partial}{\partial_t}\int_{\theta=0}^{\pi}\delta(t-\frac{Rcos\theta}{c})sin\theta d\theta \\
&= -\frac{1}{4\pi^2 R}\frac{\partial}{\partial_t}\int_{\tau=-R/c}^{R/c}\delta(t-\tau)d\tau \\
&= \frac{1}{4\pi^2 R}\left(\delta(t-\frac{R}{c}) - \delta(t+\frac{R}{c})\right).
\end{aligned} \tag{5.13}$$

The last term in Equation 5.13 is the Green kernel as it is defined in Equation 5.9. The second term is an anti-causal effect which expresses the fact that what happens now on x is related to what will happen on y at the future time $t+|x-y|/c$. This has no physical meaning and this part should be carefully removed in the algorithm using causality constrains. In the rest of the study we now use $\tilde{G}$ instead of G. We remind that we have $\vec{R} = \vec{yx} = y\vec{M} + M\vec{M'} + \vec{M'}x$ and considering that $\delta(t-t_1) * \delta(t-t_2) = \delta(t-t_1-t_2)$, this decomposition of $\vec{R}$ can be inserted directly in $\tilde{G}$ which

is transformed into several time convolution product

$$\tilde{G}(R,t) = -\frac{1}{8\pi^2 c}\int_{\vec{s}\in S}\frac{\partial}{\partial_t}\delta(t-\frac{\vec{s}.\vec{M'x}}{c}) * \delta(t-\frac{\vec{s}.M\vec{M'}}{c}) * \delta(t-\frac{\vec{s}.y\vec{M}}{c})d\vec{s}\,. \tag{5.14}$$

We follow the choice of the original study to derivate the term relative to $M'x$.

Distant Interactions (Far-Field)

In this section, we aim to compute the interaction between two cells $\mathcal{C}$ and $\mathcal{C}'$ of centers M and M' respectively. This computation includes the projection of the sub-domains $\Gamma\cap\mathcal{C}$ and $\Gamma\cap\mathcal{C}'$. The resulting equation is given by

$$s_i^m = c\int_{t\in\mathbb{R}}\int_{y\in\Gamma\cap\mathcal{C}}\int_{x\in\Gamma\cap\mathcal{C}'}\tilde{G}(|x-y|,t) * \Phi(y,t)\varphi_i(x)\chi_m(t)dxdydt\,. \tag{5.15}$$

By using Equation 5.14, Equation 5.15 becomes

$$\begin{aligned} s^m_{i,(\mathcal{C},\mathcal{C}')} = -\frac{1}{8\pi^2}\int_{t\in\mathbb{R}}\int_{\vec{s}\in S}\int_{y\in\Gamma\cap\mathcal{C}}\int_{x\in\Gamma\cap\mathcal{C}'}\frac{\partial}{\partial_t}\delta(t-\frac{\vec{s}.\vec{M'x}}{c}) * \delta(t-\frac{\vec{s}.M\vec{M'}}{c}) * \delta(t-\frac{\vec{s}.y\vec{M}}{c}) \\ *\Phi(y,t)\varphi_i(x)\chi_m(t)dxdydtd\vec{s}\,. \end{aligned} \tag{5.16}$$

By reordering the integrals, we obtain

$$\begin{aligned} s^m_{i,(\mathcal{C},\mathcal{C}')} = -\frac{1}{8\pi^2}\int_{t\in\mathbb{R}}\int_{\vec{s}\in S}\int_{x\in\Gamma\cap\mathcal{C}'}\frac{\partial}{\partial_t}\delta(t-\frac{\vec{s}.\vec{M'x}}{c}) \\ * \left[\delta(t-\frac{\vec{s}.M\vec{M'}}{c}) * \left(\int_{y\in\Gamma\cap\mathcal{C}}\delta(t-\frac{\vec{s}.y\vec{M}}{c}) * \Phi(y,t)\right)\right] \\ \varphi_i(x)\chi_m(t)dxdydtd\vec{s}\,. \end{aligned} \tag{5.17}$$

Like in the frequency FMM, Equation 5.17 leads to three computation stages which are the initialization, the transfer pass and the integration pass.

Initialization ($P2M$). In this step, we compute for each leaf $\mathcal{C}$ the radiation function $F_C(\vec{s},t)$ defined on the unit sphere by

$$F_C(\vec{s},t) = \int_{y\in\Gamma\cap\mathcal{C}}\delta(t-\frac{\vec{s}.y\vec{M}}{c}) * \Phi(y,t)dy\,. \tag{5.18}$$

Therefore, F_C is related to only Φ from Equation 5.4 and the cell $\mathcal{C}$ of center M. F_C represents the influence of $\Gamma\cap\mathcal{C}$ on the external direction.

Transfer ($M2L$). In this stage, we transfer the result from the $P2M$ on the cell $\mathcal{C}$ of center M to the cell $\mathcal{C}'$ of center M'. We perform a convolution of the function F_C by the transfer function $T_{\vec{MM'}}$

which is related to $\vec{s} \in S$ and t

$$T_{M\vec{M}'}(\vec{s},t) = \delta(t - \frac{\vec{s}.M\vec{M}'}{c}) \,. \tag{5.19}$$

The result is still a function defined on S, since it represents the impact of Φ from $\Gamma \cap \mathcal{C}$ on the point M'.

Integration ($L2P$). The last stage performs an integration of the transfer result on three components: S, the time t and $\Gamma \cap \mathcal{C}'$

$$-\frac{1}{8\pi^2} \int_{t\in\mathbb{R}} \int_{\vec{s}\in S} \int_{x\in\Gamma\cap\mathcal{C}'} \left[\frac{\partial}{\partial_t} \delta(t - \frac{\vec{s}.\vec{M'x}}{c}) * T_{M\vec{M}'} * F_C(\vec{s},t) \right] \varphi_i(x)\chi_m(t) dxdydtd\vec{s} \,. \tag{5.20}$$

We can reintroduce the normal derivatives removed in Equation 5.11. From the original study, the term $\frac{\partial}{\partial n_x}$ can be inserted into the $P2M$, and $\frac{\partial}{\partial n_y}$ in the $L2P$.

5.1.5 Unit Sphere Discretization

In the kernel decomposition, we use time-dependent functions defined on the unit sphere S in $\mathbb{R}^3$. By introducing

$$g(x,\vec{s},t) = \int_{y\in\Gamma\cap C} \delta(t - \frac{\vec{s}.\vec{M'x}}{c}) * \delta(t - \frac{\vec{s}.y\vec{M}}{c}) * \Phi(y,t) dy \,, \tag{5.21}$$

Equation 5.17 writes

$$s_i^m = -\frac{1}{8\pi^2} \int_{t\in\mathbb{R}} \int_{\vec{s}\in S} \int_{x\in\Gamma\cap C'} g(x,\vec{s},t) * \frac{\partial}{\partial_t} T_{M\vec{M}'} \varphi_i(x)\chi_m(t) dxdydtd\vec{s} \,. \tag{5.22}$$

For a given point x, the function $g(x,\vec{s},t)$ can be described as a radiation function on a domain with a radius equal to twice the radius of C. Moreover, this function from $\mathbb{L}^2(S)$ is expended in spherical harmonics function ($Y_{l,m}$) with $l \geq 0$ and $-l \leq m \leq l$. In [104; 105; 106], they study the bandwidth of the functions over the spherical harmonics to determine the bound $l < L$ which gives an accurate representation of g. From these studies, the bound L is chosen by

$$\begin{aligned} L &= k_s d + C_\varepsilon . log_{10}(k_s d + \pi) \\ &= (\omega_s/c) d + C_\varepsilon . log_{10}((\omega_s/c) d + \pi) \,. \end{aligned} \tag{5.23}$$

The term d is the diameter of the box C, see Section 5.1.3. The other terms k_s and C_ε are parametrized for the target simulation; $k_s = \omega_s/c$ is the wave number associated to the maximal frequency we aim to study and C_ε is the accuracy parameter (for example $C_\varepsilon = 7$). More details are provided regarding these two parameters in Section 5.1.6.

$T_{M\vec{M}'}$ is also a function of bandwidth L such that the product of the two functions is of bandwidth $2L$. Therefore, all the functions that we integrate have a bandwidth limited by $2L$ and are of the

form

$$\sum_{\substack{-l\leq m\leq l\\ 0\leq l\leq 2L}} A_{l,m} Y_{l,m}(\vec{s}) \,. \tag{5.24}$$

Therefore, we use the same discretization of the unit sphere as the ones from the frequency FMM. This discretization is composed by a grid (θ_i, φ_j) of $(L+1)(2L+1)$ points. The θ_i are the Gauss-Legendre integration points and the φ_j are distributed equally on the interval $(0, 2\pi)$ (we have $\varphi_j = j2\pi/(2L+1)$). In Appendix B.2, we give different discretization methods and for the rest of the study we consider the use of the advanced approach, see Appendix B.2.2.

5.1.6 APS (Approximate Prolate Spheroidal)

From the previous definitions, an appropriate interpolation function must have a limited time-support but also to be band-limited in frequency. We choose an extension of the approximate prolate spheroidal function (APS) proposed in [107] which guarantees a correct accuracy up to a a given frequency f_{max} (see [3] for a complete study). The function is given by

$$APS(t) = \frac{\omega_+}{\omega_c} \frac{sin(\omega_+ t)}{\omega_+ t} \frac{sinh(p_t \Delta t \omega_- \sqrt{1 - t^2/p_t^2 \Delta t^2})}{sinh(p_t \Delta t \omega_-)\sqrt{1 - t^2/p_t^2 \Delta t^2}} \,. \tag{5.25}$$

The function APS is parametrized by p_t and χ_s, but it is also linked to variables from the problem definitions and the choice of the frequency study (the variables ω_+ and ω_-. are defined further in this section). We describe these different variables and their relations. From the simulation properties, we have Δt and Δx which give us the ratio $n_{CFL} = \frac{c\Delta t}{\Delta x}$. Usually, to ensure that from one time step to the other the waves do not propagate to fast on the mesh, we must have $n_{CFL} < 1$ and it is usual to choose $n_{CFL} = 1/2$. However, in this formulation, any value of n_{CFL} would work, since the mathematical scheme is unconditionally stable, see [2].

The FMM method is parametrized by a maximum frequency f_{max} (chosen as is the maximum frequency to study). Associated to f_{max}, we have λ_{max} the wavelength, $\omega_{max} = 2\pi f_{max}$ the pulsation associated to f_{max} and $\frac{\lambda_{max}}{\Delta x}$ the number of points per wavelength (in space). It is frequent to define the maximum usable frequency on the mesh as $f_{max} = c/10\Delta x$ ($\omega_{max} = 2\pi c/10\Delta x$) which corresponds to 10 points per wavelength λ_{max}.

The parameters p_t and χ_s used in the function APS are also connected with the target accuracy ε_t by

$$\begin{aligned} \varepsilon_t &\leq \frac{1}{sinh(p_t \Delta t \omega_-)} \\ &\leq \frac{1}{sinh(p_t \frac{\pi}{2} \frac{\chi_s - 1}{\chi_0})} \\ &\leq \frac{1}{sinh(p_t \frac{\pi}{2} \frac{(\omega_s/\omega_{max}) - 1}{(\omega_0/\omega_{max})})} \end{aligned} \tag{5.26}$$

Therefore, for a chosen accuracy ε_t and parameter p_t, the minimal χ_s is given by

$$\chi_s = (\frac{2\chi_0}{p_t\pi} sinh^{-1}(\frac{1}{\varepsilon_t})) + 1 = (\frac{1}{\Delta t f_{max} p_t \pi} sinh^{-1}(\frac{1}{\varepsilon_t})) + 1\,. \tag{5.27}$$

The number of discretization points over the unit sphere is based on the parameter L which itself is connected to k_s (or χ_s), see Section 5.1.5. On the other hand, the APS function has non zero values in the interval $[-p_t\Delta t, p_t\Delta t]$, and after a time discretization, the signals are of length $2p_t - 1$ values. Therefore, for a given accuracy, p_t is the parameter that allows to balance between the number of discretization points of the unit sphere and the length of the signals from the function *APS*. It should be carefully chosen as we show in the study part in Section 5.5.3.

We summarize here some of the notations we used in our formula but refer to the original study to have more details

- $\omega_c = \frac{\pi}{\Delta t}$: the time discretization requires that we work in band limited signals.
- $\chi_0 = \frac{\omega_c}{\omega_{max}}$: oversampling ratio (in our case $\chi_0 = 5/n_{CFL} = 10$).
- $\chi_s = \frac{\omega_s}{\omega_{max}}$: it is related to the maximum frequency and the accuracy (we choose it - or choose ω_s).
- $k_s = \omega_s/c$: Number of wave associated to the maximal frequency.
- $\omega_{max} < |\omega| < \omega_c$: The pulsation over ω_{max} are not study and will certainly be wrong.
- $|\omega| < \omega_{max}$: this interval represent the pulsations we want to study (which are lower than ω_{max}).
- $\omega_c < |\omega|$: pulsation that should never exist into the computation to ensure correct results.
- $\omega_+ = (\omega_s + \omega_{max})/2$: relation used in the r (APS) function.
- $\omega_- = (\omega_s - \omega_{max})/2$: relation used in the r (APS) function.

The shape of the function r is shown in 5.6.

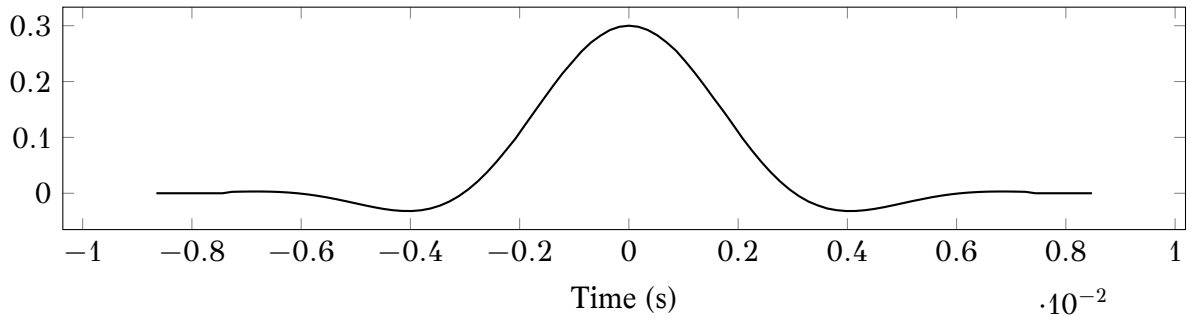


Figure 5.6: APS function shape for $\Delta t = 9.1 \cdot 10^{-4}, p_t = 8, \omega_{max} = 3.45 \cdot 10^2, \omega_c = 3.45 \cdot 10^3$ and $\omega_s = 1.72 \cdot 10^3$.

5.2 Operators

In this section, we describe the FMM operators as we implement them in our application. We remind that, from the formulation, we work with functions defined on the unit sphere which represent the emission of a cell/box to the outside.

5.2.1 P2M

The $P2M$ computes a Gaussian quadrature over the unknowns and transfers the values with the APS function to the Gauss points from the unit sphere discretization. In the first stage, we evaluate the source Gauss points (G_k) that are located inside a leaf with a Gaussian quadrature

$$\vec{h}(G_k) = p_k n_{\vec{G_k}} \sum_{n_{dl}^C}^{j=1} \lambda_j \varphi_j(G_k)\,. \tag{5.28}$$

In Equation 5.28, $\varphi_j(G_k)$ is the evaluation of the unknown j at gauss point k.

In a second step, we transfer the potentials of the source Gauss points to the unit sphere Gauss points of the leaf $\mathcal{C}$ of center M. In the time-domain, the scalar potential is transformed into an APS signal which is added to each Gauss point of the unit sphere with the appropriate time shift and magnitude. The formula is given by

$$F_C(\vec{s},t) = \left[\sum_{k=1}^{n_{ptG}^C} r(t - \frac{\vec{s}\cdot\vec{G_kM}}{c} - t_n)\vec{h}(G_k)\right].\vec{s}\,. \tag{5.29}$$

Each Gauss point represents an outer direction $\vec{s}$ on the unit sphere and stores a discretized signal. More precisely, the APS function is non zero on the time step interval $[-p_t, p_t]$ and the maximum delayed between a source Gauss point G_k and the sphere that includes the leaf is r_t time steps such that the signals are non zero on $[-p_t - r_t, p_t + r_t]$. These time-domain signals can be transformed into frequency-domains signals with a Fourier transform but it is also possible to perform this transformation directly in the $P2M$ by

$$\widetilde{F}_C(\vec{s},\omega) = \left[\sum_{k=1}^{n_{ptG}^C} \widetilde{r}(\omega) e^{-i\frac{\vec{s}\cdot\vec{G_kM}}{c}\omega}\vec{h}(G_k)\right].\vec{s}\,. \tag{5.30}$$

The interest to choose the time-domain or the frequency-domain depends on the working domain of the next operators.

Flops Evaluation

The first step is independent from the target accuracy and the same computation is performed for the time-domain or the frequency-domain $P2M$. Let n_C^{dl} the number of unknowns in a leaf C, $n_{average}^G$ the average number of unknowns per Gauss point and $G_F = G/F$ the average number of Gauss points in each leaf. The number of Flop is given by $F_{U-to-G} = G_F \times n_{average}^G \times 14$, where 14 is the number of operations to evaluate an unknown. The complexity of this operation is linear relatively to N since there is a linear relation between the number of unknowns and the number of Gauss points.

The second step iterates on the leaves and performs a triple loop on each of them. The first loop is over the Gauss points inside each leaf (n_C^{dl}), the second loop is over the Gauss points of the

unit sphere discretization ($(L+1)\times(2L+1)$), and the third loop is over the time steps of the APS signal ($2\times(p_t+r_t)-1$). The total number of Flop per leaf in the time-domain is given by $F_{P2M}^{time-domain} = (L+1)\times(2L+1)\times n_{average}^{G}\times((2\times(p_t+r_t)-1)\times 15+10)$. In this stage the target accuracy changes the parameters L and p_t which are related to the number of points in the unit sphere and the length of the APS signal respectively. There are limited differences between the computation in the time-domain or in frequency-domain but in frequency-domain we perform complex multiplications.

5.2.2 M2M

The $M2M$ operator converts functions from level $l+1$ into functions of upper level l. It transforms the signals defined over the the Gauss points from the unit sphere discretization of level $l+1$ and centered in M^{l+1} into signals on the Gauss points from the unit sphere discretization of level l and centered in M^{l}. The number of Gauss points per sphere grows as we go up in the tree but it depends on the accuracy which might be parametrized differently between levels. To shift the center of a sphere, we have to shift the signals in time based on the wave velocity and the distance between M^{l+1} and M^{l}. Transforming the signals of the $(L^{l+1}+1).(2L^{l+1}+1)$ points to the $(L^{l}+1).(2L^{l}+1)$ points is called the *extrapolation*. We define these two operations that compose the $M2M$ in Figure 5.7. It is possible to extrapolate before to shift or the inverse, but the accuracy is expected to be better by first extrapolate and then shift.

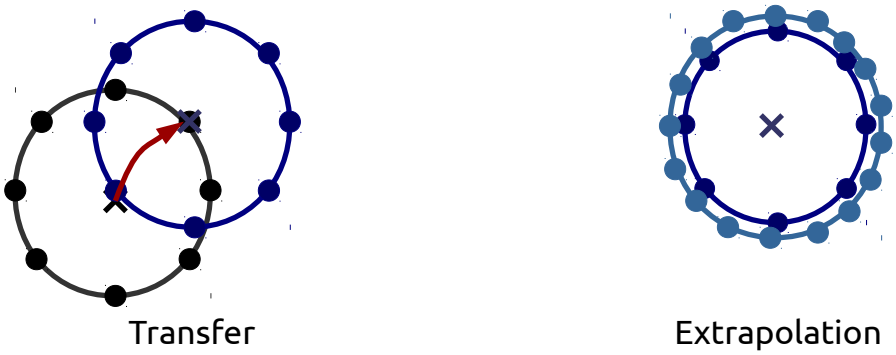


Figure 5.7: $M2M$ the time-domain BEM composed by a transfer (time shift) and an extrapolation (increasing number of Gauss points).

Time shift

The transfer between the two levels is done by shifting the signals in time relatively to the distance between the parent and the child $M^{(l+1)}\vec{M}^{(l)}$. The signals from the children are accumulated into the parent and in the time-domain we have

$$F_{C^{(l)}}(\vec{s},t) = \sum_{C^{(l+1)}\in child(C^{(l)})} \delta(t-\frac{\vec{s}\cdot M^{(l+1)}\vec{M}^{(l)}}{c}) * F_{C^{(l+1)}}(\vec{s},t)\,. \tag{5.31}$$

If the shift coefficient is a multiple of Δt the operation is a simple copy. However, the delay is usually not a multiple of the time step and we have to approximate the source signal to perform the time shift.

A time shift in the time-domain is equivalent to a phase change in the frequency-domain which is done by a complex multiplication. The frequency-domain equivalent of the Equation 5.31 is

$$\widetilde{F}_{C^{(l)}}(\vec{s},\omega) = \sum_{C^{(l+1)} \in child(C^{(l)})} e^{(-i\frac{\vec{s}\cdot \vec{M^{(l+1)}M^{(l)}}}{c}\omega)} \widetilde{F}_{C^{(l+1)}}(\vec{s},\omega)\,. \tag{5.32}$$

The frequency-domain expression hides a critical aspect of the time shift in this regime; when we change the phase of a frequency signal the time signal is rotated. Therefore, if we rotate the result of the $P2M$ without being cautious on the time window the result will be wrong. As an example, if we shift the non zero signal *abcd* by one time step in the frequency-domain, we will obtain the time signal *bcda*. To have a correct shifting, the original time window should be increased to have 00*abcd*00 and to rotate the zero values 0*abcd*000. But we cannot increase the time window and pad with zeros a frequency signal without a reverse transformation in the time-domain because it changes the pulsations. At the time of writing, no fast methods exists to change the pulsations of a frequency signal. While a time shift in the frequency-domain has a good accuracy and is not intrinsically expensive, the problem of the time window/pulsations is an important issue. On the other hand, shifting in the time-domain is done by interpolation and we have to use advanced methods to have a good accuracy, see results in Section 5.5.2.

After the $P2M$, the signals are non zero in the time interval $[-p_t - r_t, p_t + r_t]$. During the $M2M$ the signals are shifted by $\frac{\vec{s}\cdot\vec{M^{(l+1)}M^{(l)}}}{c}$ which is included in the interval $[\frac{-\vec{M^{(l+1)}M^{(l)}}}{c}; \frac{\vec{M^{(l+1)}M^{(l)}}}{c}]$. The non zero interval is increased by half the box diagonal divided by the wave velocity, r_t, the new non zero interval is $[-p_t - r_t - r_t; p_t + r_t + r_t]$.

Extrapolation

The extrapolation step is the same in the time-domain or the frequency-domain. We summarize three methods with different costs and implementation difficulties but they all convert the values of the Gauss points at level $l+1$ to the Gauss points at level l. Therefore, the conversion should be applied on all the non zero time interval in the time-domain and on all the pulsations in the frequency-domain.

Method 1: direct relation matrix. In this approach, we compute a transfer matrix $C^{l+1\to l}$ of dimension $(L^{l+1}+1).(2L^{l+1}+1) \times (L^l+1).(2L^l+1)$ which contains all the coefficients to transform the values at the unit sphere discretization Gauss points from level $l-1$ to the ones at level l. The same matrices are used for the $M2M$ and the $L2L$ and we need one matrix per level but the memory occupancy of this approach is huge because the matrix are of order $O((L^{l+1})^4)$. The main advantage of this approach is its straightforward expression and the value at position (k,k') is given

by

$$
\begin{aligned}
M_{k,k'} &= \sum_{-l\leq m\leq l 0\leq l\leq L^{(L)}} Y^{*}_{l,m}(\vec{s_k}^{(l)}) Y_{l,m}(\vec{s_{k'}}^{(l-1)}) \\
&= \sum_{0\leq l\leq L^{(L)}} \frac{2l+1}{4\pi} P_l(cos\theta)\,, \\
\text{where } cos\theta &= \vec{s_k}^{(l)}.\vec{s_{k'}}^{(l-1)}\,.
\end{aligned}
\tag{5.33}
$$

The matrix $C^{l+1\to l}$ is used either with a matrix-vector product if we work on one time interval/pulsation at a time or with a matrix-matrix product if we convert all the values from a cell. It is possible to reduce the memory footprint of the matrix by removing the values lower than a given threshold to obtain a sparse matrix.

Method 2 : Q^l_m polynomial. An alternative is to use the Q^l_m polynomial for which we provide the recurrence formula in Appendix B.1.3. The extrapolation is done by

$$
\begin{aligned}
\tilde{F}^{(l+1)}_{i,m} &= \sum_{0\leq j\leq 2L^{(l+1)}} e^{-im\varphi^{(l+1)}_j} F^{(l+1)}_{i,j}\,, -L^{(l+1)} \leq m \leq L^{(l+1)} \\
\widetilde{F}^{(l+1)}_{i,m} &= \sum_i \left[\sum_{|m|\leq l\leq L^{(l+1)}} Q^m_l(cos\theta^{(l+1)}_i) Q^m_l(cos\theta^{(l)}_{i'}) \right] \widetilde{F}^{(l+1)}_{i,j} \\
F^{(l)}_{i',j'} &= \sum_{-L^{(l+1)}\leq m\leq L^{(l+1)}} e^{im\varphi^{(l)}_j} \tilde{F}^{(l)}_{i',m}\,.
\end{aligned}
\tag{5.34}
$$

The first loop weights the $(L^{l+1}+1).(2L^{l+1}+1)$ source Gauss points using the values from the unit sphere discretization. In a second step, we apply a DFT to the Gauss points in the φ direction. Then we convert the sources Gauss points to the target Gauss points using the Q^l_m polynomial. The last step applies an inverse DFT to be back in the original domain. These different stages must be done for all the non zero time steps/pulsations of the input vectors. The complexity is close to the direct method but the memory occupancy is much lower because we do not store the complete C matrix.

Method 3: dissociated Q^l_m. This approach uses the relation

$$
\begin{aligned}
\sum_{|m|\leq l\leq L^{(l+1)}} Q^m_l(cos\theta^{(l+1)}_i) Q^m_l(cos\theta^{(l)}_{i'}) &= \sqrt{\frac{(L^{(l)+1})^2 - m^2}{4(L^{(l)}+1)^2 - 1}} \\
&\times \left[\frac{Q^m_{L^{(l)}+1}(x') Q^m_{L^{(l)}}(x)}{x'-x} - \frac{Q^m_{L^{(l)}+1}(x) Q^m_{L^{(l)}}(x')}{x'-x} \right]\,.
\end{aligned}
\tag{5.35}
$$

We apply the same operations as the Q^l_m polynomial approach but we split the computation in

three parts where the central operation is given by the matrix

$$C_{i',i} = \frac{1}{cos\theta_{i'}^{l-1} - cos\theta_i^l} \,. \tag{5.36}$$

Flop Evaluation

In our implementation of the $M2M$, we shift the signals in time and then extrapolate. Therefore, the time shift is applied on the $(L^l+1).(2L^l+1)$ Gauss points of the child unit sphere discretization. Shifting in the time-domain costs $(L^{h-1}+1).(2L^{h-1}+1).(2(p_t+r_t)-1).18$ from $h-1$ to $h-2$ because the signals are of length $(2(p_t+r_t)-1)$ and the cubic interpolation costs 18 Flop per value. In the frequency-domain, the phase shift is done by a complex multiplication which leads to a total of $(L^{h-1}+1).(2L^{h-1}+1).(2(p_t+r_t)-1).8$. The extrapolation looks similar in the frequency and time domains but in the frequency-domain we deal with complex values. When using the Q_m^l and the Q_m^l Dissociated approaches in the frequency-domain, the DFT and inverse DFT are done from complex to complex. There is a total of L^{h-1} DFT on signals of length $2L^{h-1}$ for each of the $2(p_t+r_t)-1$ different non zero time steps/pulsations. Once the polynomial has been used, there is a total of $2L^{h-2}+1$ inverse DFT on signals of length $2L^h$ on the same non zero time steps/pulsations. In addition, in any domain, the core part of the extrapolation performs $(L^{h-1}+1).(2L^{h-1}+1).(L^{h-2}+1).(2L^{h-2}+1)$ complex multiplications.

5.2.3 M2L

The transfer is done by a convolution product with the function

$$\begin{aligned} T^L_{M\vec{M'}}(\vec{s},t) &= \frac{\partial^2}{\partial t^2}\delta(t - \frac{\vec{s}.M\vec{M'}}{c}) \,, \\ G_{C'}(\vec{s},t) &= T^L_{M\vec{M'}}(\vec{s},t) * F_C(\vec{s},t) \,. \end{aligned} \tag{5.37}$$

By considering the spherical harmonics of rank lower or equal than L, $T^L_{M\vec{M'}}$ is approximated by

$$T^L_{M\vec{M'}}(\vec{s},t) = \begin{cases} \frac{c}{MM'}\frac{\partial^2}{\partial t^2}\sum_{l=0}^{L}(l+1/2)P_l\left(\frac{ct}{MM'}\right)P_l\left(cos(\vec{s},M\vec{M'})\right) \,, \; |t| < \frac{MM'}{c} \\ 0 \,, otherwise \end{cases} \tag{5.38}$$

There is a finite number of non zero values based on the distance MM' and the time discretization step Δt. The function $T^L_{M\vec{M'}}$ is of length $2d_t-1$, thus the convolution product extend the length of non zero interval of the source vectors by $2d_t-2$, where $d_t = round\_down(\frac{MM'}{c})$. The interactions of the $M2L$ are from different distance relatively to the target cell, such that the transfer signals have different non zero length.

A convolution product in the time-domain is equivalent to term-by-term multiplications in the frequency-domain. The transfer function $\tilde{T}^L_{M\vec{M'}}(\vec{s},\omega)$ must be computed directly in the frequency-domain because it is not band limited, and computing the DFT of $T^L_{M\vec{M'}}(\vec{s},t)$ results in an incorrect

signal. The transfer signal in the frequency-domain is given by

$$\begin{aligned}\tilde{T}^{L}_{M\vec{M}'}(\vec{s},\omega) &= \sum_{l=0}^{L}(2l+1)(-i)^l j_l\left(\frac{\omega MM'}{c}\right)P_l\left(cos(\vec{s},M\vec{M}')\right)\\ \tilde{G}_{C'}(\vec{s},\omega) &= \tilde{T}^{L}_{M\vec{M}'}(\vec{s},\omega).\tilde{F}_C(\vec{s},\omega)\,.\end{aligned} \tag{5.39}$$

Like the other operators, the $M2L$ can be computed either in the time or the frequency domain using the appropriate conversion. For example, if the $M2M$ is done in the time-domain, we transform the multipole part of the source cell in the frequency-domain by a DFT, and then we can use the $M2L$ in the frequency domain. Therefore, we use the $M2L$ in the frequency-domain even if all the other operators are in the time domain.

We provide the recurrence formulas to compute the Legendre polynomials and its derivative in Appendix B.1.1 and Appendix B.4.1 respectively.

Flop Evaluation

We do not count the construction of the matrices which is perform during a pre-computation stage. We remind in Appendix B.3.4 that the result of a discrete convolution product between two signals of lengths l_A and l_B is of size $l_{A*B} = l_A + l_B - 1$ with a complexity of $O(l_A \times l_B)$. Therefore, a single $M2L$ interaction costs $(L^l+1).(2L^l+1) \times (2r_t-1) \times (2p_t-1)$ Flop in the time-domain. In the frequency-domain, the term-by-term multiplications lead to $(L^l+1).(2L^l+1) \times 4(p_t-1) \times 6$ Flop, considering that a complex multiplication costs 6 Flop.

5.2.4 L2L

The $L2L$ operator is similar to the $M2M$ but instead of an extrapolation it uses an interpolation. In the interpolation, we convert the $(L^l+1).(2L^l+1)$ Gauss points to the lower level discretization with $(L^{l-1}+1).(2L^{l-1}+1)$ Gauss points, and with $L^l \geq L^{l-1}$. The interpolation is the transposed of the extrapolation and thus the same methods are applied. As in the $M2M$ there is a time shift based on the distance between the parent cell and its children. The number of Flop for a $L2L$ is the same as the $M2M$ except that the length of the non zero time interval is much greater because it has been extended by the $M2L$ at the same level and particularly by the upper $M2L$ propagated by the upper $L2L$.

5.2.5 L2P

After the $M2L$ at the leaf level and the $L2L$ from $h-2$ to $h-1$, we have $(L^{h-1}+1).(2L^{h-1}+1)$ time-domain signals per leaf. The TD BEM algorithm needs the summation vector s^n and thus some of the values from these signals must be applied to the unknowns. These time signals cover several time steps and we might not need all their values to obtain the current time step summation vector. Therefore, we just apply the values that will modify the current summation vector s^n (but also the future summations vectors $s^{p>n}$ in case of incomplete FMM as presented in Section 5.3.2).

Conversely to the $P2M$, we first transfer from the unit sphere discretization Gauss points to the leaves Gauss points and then to the unknowns. The first transfer is done by

$$\vec{g}(G_k,t) = \int_{\vec{s}\in S} \mathfrak{G}_{C'}(\vec{s}, t - \frac{\vec{s}\cdot\vec{M'G_k}}{c})\vec{s}d\vec{s}\,. \tag{5.40}$$

Then the transfer from the Gauss points to the unknowns is done by

$$\Lambda_i^m = \frac{1}{8\pi^2 c^2}\sum_{k=1}^{n_{ptG}^{C'}} \vec{n}_{G_k} p_k \varphi_i(G_k)\left[\vec{g}(G_k,t_m) - \vec{g}(G_k,t_{m-1})\right]\,. \tag{5.41}$$

These operations must be done in the time-domain because we want to end with signals in time-domain that are compatible with our TD-BEM.

Flop Evaluation

The first step is done with a double loop over the Gauss points from the unit sphere discretization and the leaf Gauss points to shift the time vectors. The Flop cost is given by $(L+1)\times(2L+1)\times n_{average}^G \times v \times 18$ with v the length of the non zero time signals depending on the operations that were applied to the resulting local part of the leaves. The transfer from the Gauss points to the unknowns in a double loop of cost $G_F \times n_{average}^G \times v$.

5.3 Optimizations

5.3.1 Discussion on the Choice of Time or Frequency Operators

The different FMM operators presented in Section 5.1 are formulated in the frequency-domain or the time-domain. In addition, the domain can be different between the operators, between the levels or from one iteration to next because we can pass from one domain to the other with a Fourier transform. The time shift is the central operation which makes its computation critical for the performance. We remind that a time shift in the time-domain is done with an interpolation as the ones presented in Appendix B.5, whereas in the frequency-domain we introduce a phase shift in the signals with complex multiplications. The presented techniques have a linear complexity but different accuracies that we study in the experimental Section 5.5.

In Section 5.2.2 we introduce the drawback of working in the frequency-domain when we have to shift the signal in time. We remind that the extremities of the signal should be zero to ensure a correct phase shift which rotate the time values. As an example, if we transform a time signal of N non-zero values into a frequency-domain signal using a DFT, we end up with N complex numbers (or $N/2$ since there is a symmetric relation between the complex values).This frequency signal cannot be shifted in time by any coefficient otherwise some values in the time signal will rotate from the front to the back. If we want to shift this signal by -1.5 time steps, we must add two zeros in front of the time signal before transforming it in the time-domain with a DFT and we end-up with $N+2$ complex numbers and different pulsations. Using the operators in the

frequency-domain, implies to change the time window by transforming the frequency signals in the time-domain, pad with enough zeros and then transform back in the frequency-domain before apply the phase shift. This adds one DFT and one inverse DFT to the cost of the phase shift. An alternative is to find the maximum possible shift in our entire FMM to pad the signals in the $P2M$ with enough zero to ensure that all the shifts will be correct. The extra-cost of this method is a huge memory occupancy but also much more computation because many operators have the length of the vector in their complexity. In other words, the time shift in the frequency-domain involves extra DFTs or additional memory occupancy and computation.

5.3.2 Incomplete FMM

In the matrix computation presented in Chapter 3, the state of an unknown i is stored in $a^n(i)$ and is used with the different interaction matrices depending on the distance between i and the other unknowns. If the distance between the unknowns i and j is around k time steps, the value $a^n(i)$ is used at time step $n+k$ with the matrix M^k. The FMM has a similar property and the distance between the leaves and the successive time shifts change the date of usage of the values.

The $P2M$ generates non zero values on the time interval $[-p_t - r_t, p_t + r_t]$. These signals are shifted by the $M2M$, $M2L$ and $L2L$ before being applied by the $L2P$. In the $M2L$ at level l, the minimum distance between two interacting cells is the width of a single box w_l. The radius of a covering sphere at this level (R_l) reduces this distance to $o^l = w_l - 2(R_l - w_l)$. It means that the value at time zero in the source cell will not be used before $\tilde{o}^l = (o^l/(c\Delta t)) - p_t - r_t$ steps because the $M2L$ will delay the signal. The signals from the $P2M$ are not needed before at least $\tilde{o}^l$ time steps.We keep the values in the leaves but we shift them by one time step during $\tilde{o}^l$ iterations which increases the non zero interval to $[-p_t - r_t - \tilde{o}^l, p_t + r_t]$. Then we must apply the $M2L$ because some values in the leaves will be used for the current summation vector. This propagation allows us to compute an incomplete FMM for several iterations; a complete FMM stands for a complete upward, transfer and downward passes.

While this reduces the amount of work, this also leads to a more complex algorithm. In terms of implementation, we round the $\tilde{o}^l$ coefficient to a multiple of the lower coefficient $\tilde{o}^{l+1}$. If we have $\tilde{o}^{h-1} = 2$, we store on the cells of the leaf level, the results of two iterations. Each two iteration, we compute the $M2L$ at the leaf level. It is more convenient not to keep the transfered values at the leaf level because we use this memory emplacement for the third iteration $P2M$. We solve this problem by computing the $M2M$ between $h-1$ and $h-2$, and more generally we apply the $M2M$ from $h-1$ to $h-2$ each $k \times \tilde{o}^{h-1}$ time step. Therefore, at $h-2$ it is natural to compute the transfer pass at a multiple of $\tilde{o}^{h-1}$. Moreover, by doing so we compute a single downward pass which takes into account all the transfers which have been done at a given step.

5.3.3 ScalFMM as Parallelization Engine

Our time-domain BEM FMM kernel has been developed above ScalFMM following its kernel API/interface. We benefit of the ScalFMM parallelization strategies as presented in Chapter 4.

However, ScalFMM has been originally developed for particle simulations and some parameters should be customized to manage the fact that the cost of our operators increases as we go up in the tree (which is not the case with usual particle-wises kernels). For example, the chunk size of the classic fork-join division might need to be reduced. Moreover, there is a maximum of 64 cells at level 2 and thus a level-by-level approach is not able to involve more than 64 threads. In addition, the default balancing techniques for the Hybrid OpenMP/MPI parallelization are done at the leaf level using the number of leaves or number of particles per leaf.

5.4 Expression of the $4D$ FMM

One can describe our time-domain BEM FMM as a $4D$ FMM because it involves the spatial dimension but also the time dimension. The interactions between elements take into account the distance, which leads to different paths in the FMM tree, and the time which delayed the operations. In the complete FMM algorithm, we define the dependencies between the operation with the *tasks-and-dependencies* strategy for example. We can imagine a similar approach to take into account the time dependencies.

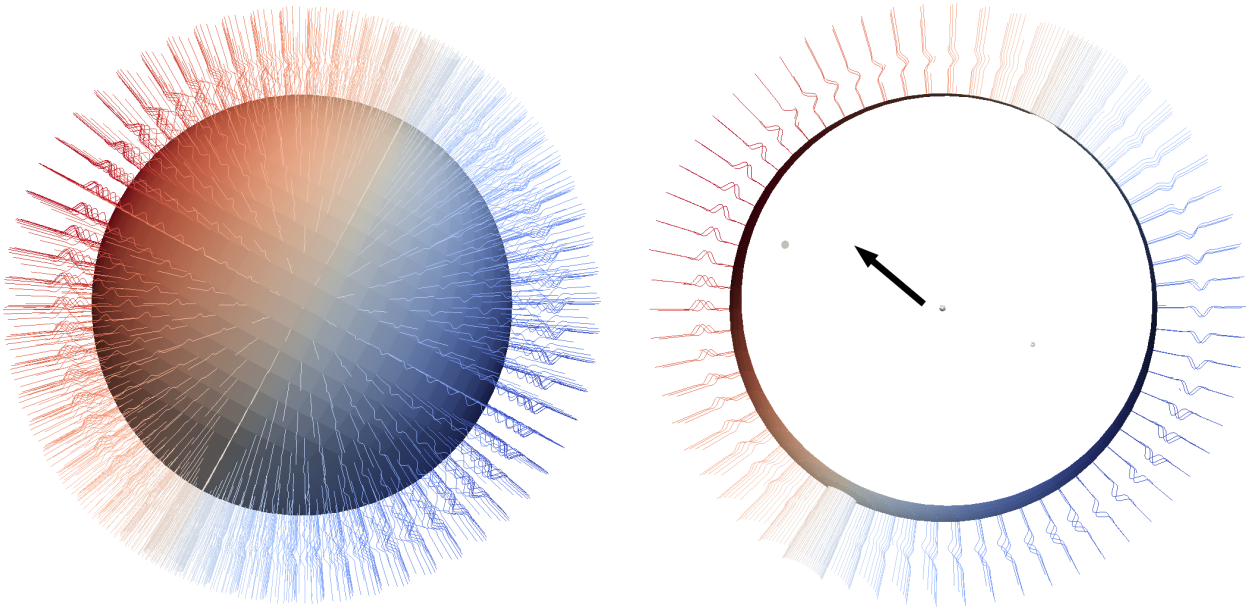
The straightforward implementation should take into account the fact that we do not need a complete FMM at each iteration by inserting the tasks up to a given level. A more finer implementation should express the time dependency between the cells and the operations. However, the time shifts are not multiples of Δt which makes the identification of the time dependencies difficult to find. The scalar states of the unknowns are transformed into a discrete signals and it is not easy to found out which values inside this vector is needed by a remote leaf. Moreover, the time shift operation in the time-domain needs a time window for each interpolated value (to use Taylors expansion for example) and thus the operators cannot easily be divided in the time direction. The same problem is encountered in the frequency-domain because the signals are in form of pulsations and cannot be spitted in the time direction. In addition, the current runtime systems do not provide hierarchical data dependencies which would be needed to fully express the dependencies in time. However, this idea might become achievable and competitive with new hardware or new *tasks-and-dependencies* expressions.

5.5 Preliminary Numerical Results and Parallel Study

5.5.1 Representation of the Unit Sphere

In Figure 5.8, we represent the unit sphere discretization of a leaf after a $P2M$. In the corresponding case, a single unknown is emitting in the upper-left direction. We remind that from a scalar value, the $P2M$ creates a signal with several values in time. On the figure, we see each signal of the Gauss points of the sphere and we recognize the APS signal. Depending on the Gauss point orientation/directions and distances relatively to the emitter, the signals are more or less shifted and with different magnitudes.

In Figure 5.9, instead of representing the signals on the Gauss points, we represent the different



(a) Complete unit sphere

(b) Truncated unit sphere

Figure 5.8: Unit sphere example at leaf level with an emission on the upper-left direction. The time signals attached to each Gauss point are represented and we recognize the APS function. The magnitude of the signals are represented from blue (low) to red (high).

time steps separately. Each sphere represents the magnitude of the signals for the different Gauss points at a given time step. This example shows a unit sphere at the leaf level which receives a contribution from its front and thus we see the propagation of the wave to the back.

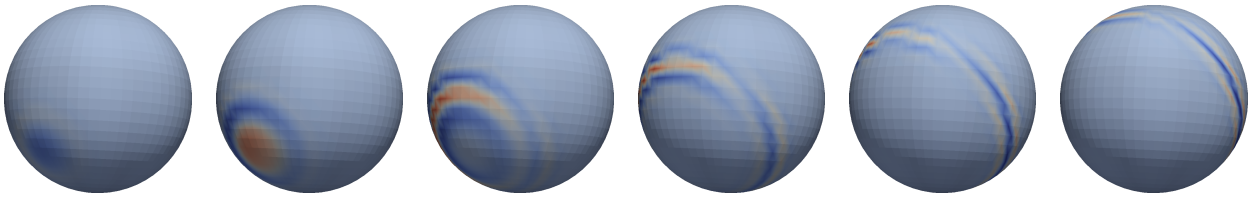
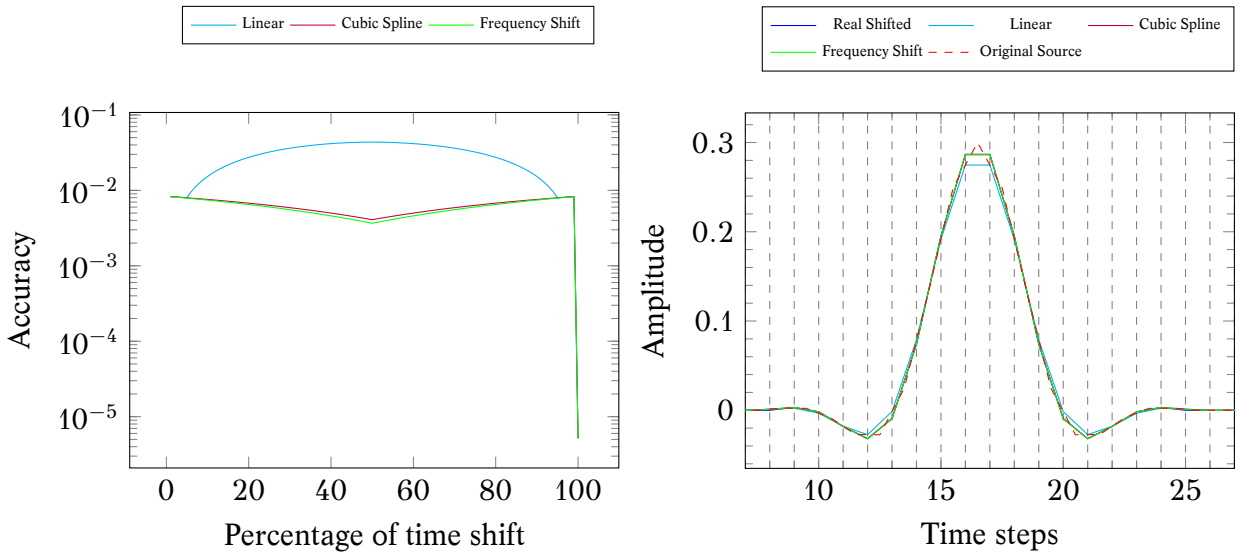


Figure 5.9: Propagation of the wave for several time steps on a target discretized sphere. The different spheres represent the values that will be applied on the included mesh elements.

5.5.2 Time Shift Evaluation

We discuss in Section 5.3.1 how critical is the time shift operation for the accuracy and the performance. In Appendix B.5, we explain how we use the linear interpolation, the cubic interpolation or a frequency phase shift to perform a time shift. In Figure 5.10, we compare the accuracy of these three methods to shift an APS function against the real APS function. We see that the linear-interpolation shift is less accurate but that the two other methods are very close. It means that there is no advantage in terms of accuracy to use the frequency-domain shift. Therefore, the choice between these two methods is only driven by the performance. We remind that the shift in the frequency-domain requires to play with the time window, which costs numerous DFT and inverse DFT.

Efficient libraries exist to perform the Discrete Fast Fourier Transform, and the two more



(a) Time Shift Accuracy for $t \in (0; \Delta t)$

(b) Time Shift Example $(0.5\Delta t)$

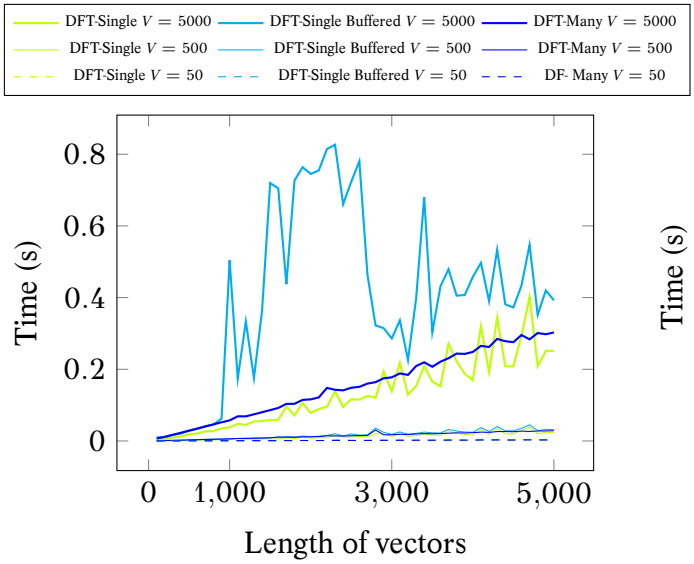
Figure 5.10: Time shifting accuracy for an APS function. The shift is done for $t \in (0; \Delta t)$. The accuray for $t = k\Delta t$ is equal to the machin precision since it is just a move of the values in the array.

widespread are the FFTW library [108] and the Intel MKL [47]. In our application, we have to compute multiple DFT of the same size because all the vectors of a cell have the same length. In Figure 5.11, we show the time to compute several DFT for two pairs of hardware/library configurations. We see that computing each DFT individually (DFT-Single by *fftw_plan_dft_1d*) is faster than computing the DFT for all vectors in a single call (DFT-Many by *fftw_plan_many_dft*). The difference is even more important for the MKL. In the DFT-Single we use a single result vector for each DFT successively, whereas for the DFT-Single Buffered we use V distinct buffer. Therefore, the difference between the two methods comes from memory effect and we see that it has an important impact on the *i*7.

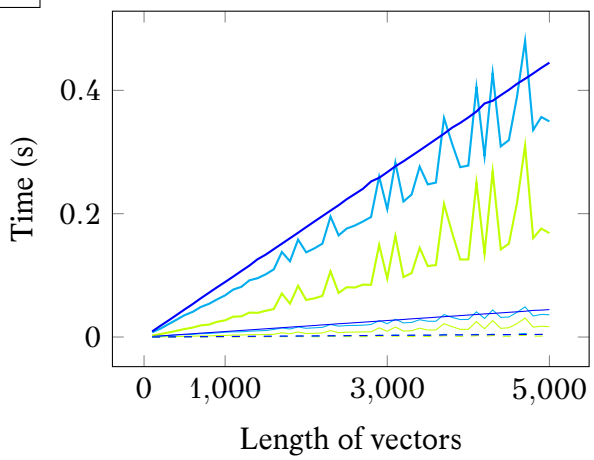
We remind that a time shift in the frequency-domain implies to increase the time window (one DFT plus one inverse DFT per vector) and to multiply each complex by a complex value. In Figure 5.12, we present the time taken to perform the time shift in the frequency-domain and in the time-domain using a cubic interpolation where the derivative are based on Taylor expansions. For the frequency-domain we count only one DFT (instead of one DFT plus one inverse DFT) but it is much slower compared to the time-domain shift. Moreover, our cubic interpolation is as accurate as the frequency-domain shift and could be optimized with SIMD intrinsics. Therefore, it looks advantageous to perform the time shift in the time-domain.

5.5.3 Parametrization

We explain in Section 5.1.6, that once the accuracy is chosen we can change two parameters p_t and χ_s. These values are tied to guarantee the target accuracy but they should be minimal to reduce the workload. For a given case, we look at the relation between those two parameters in Figure 5.13a. The *sinh* function in the original formula makes the χ_s becoming very large as we reduce p_t. We recall that χ_s is used to find the parameter L which is used in the sphere discretization, whereas p_t



(a) Time to compute all the DFT for the V vectors on a Intel $i7-4610M$ at $3.00GHz$ and with FFTW $fftw-3.3.4$.



(b) Time to compute all the DFT for the V vectors on a Intel Xeon $E5-2680$ at $2,50GHz$ and with Intel MKL 11.2 (2015.3.187).

Figure 5.11: FFTW Performance for three set of vectors (V) and different vector lengths.

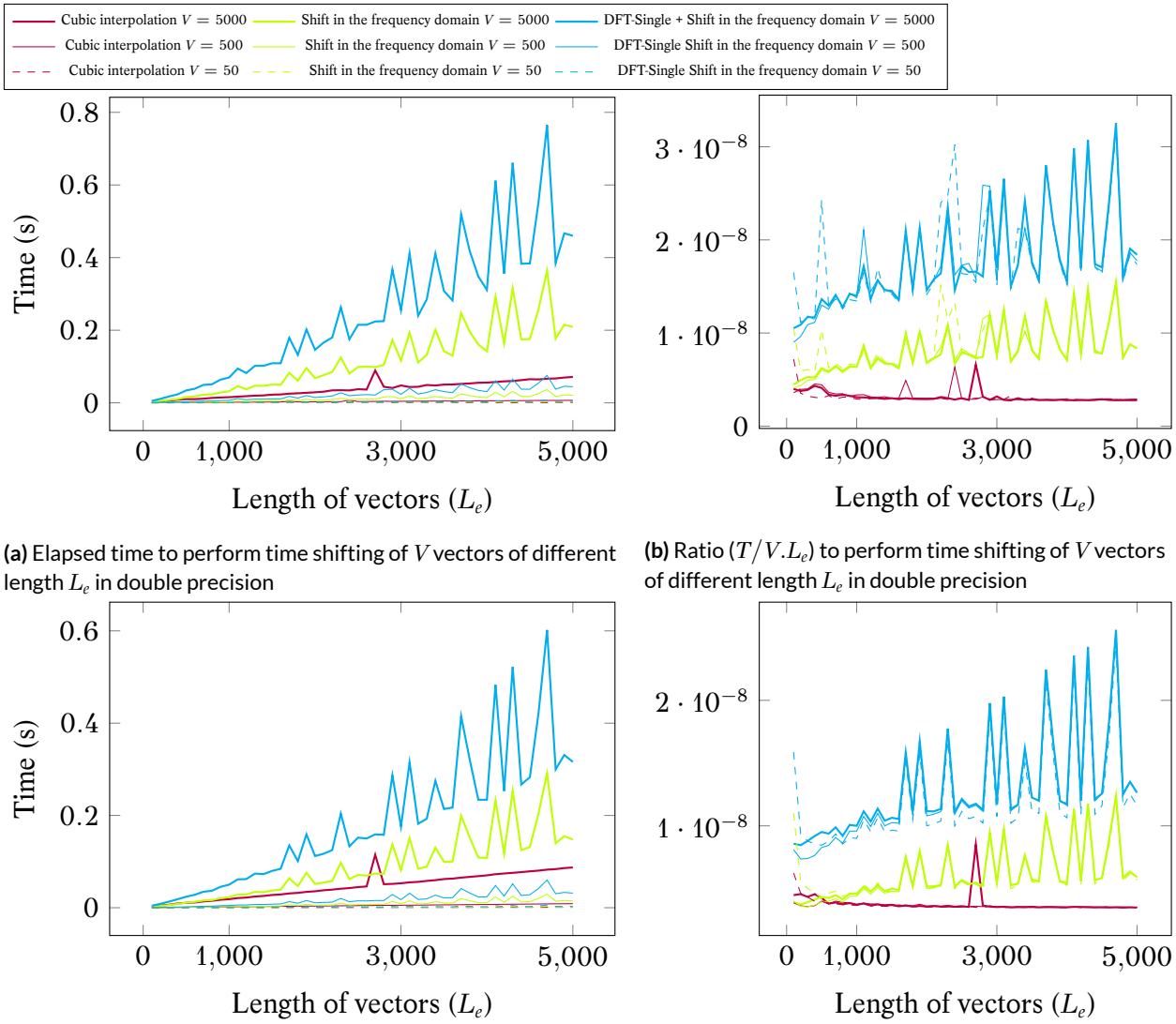
is used in the length (V) of the time-domain signals on the Gauss points from the unit sphere discretization. Figure 5.13b shows the resulting coefficients L and V for different p_t, and considering that we use the lowest χ_s. We can estimate the cost of the operators by looking at the values each cell contains. In Figures 5.13c and 5.13d we show the number of values a leaf holds for two widths (W). For the target accuracy 10^{-2} the choice of p_t is crucial and a value between 10 and 20 seems an appropriate choice. In our implementation, the choice of p_t is static, but it would be possible to pre-compute the best parameters before executing the FMM simulation.

5.5.4 Test Cases

To study our implementation of the TD-FMM, we use the different test cases introduced in Section 1.2.4 based on the cone-sphere mesh. The configurations are summarized in Table 5.1 where we add some information regarding the FMM resulting octree.

Case	C-927	C-4269	C-10012	C-22468
Number of unknowns	927	4269	10012	C-22468
FMM tree height	3	4	5	6
Number of leaves in the FMM tree	16	64	234	936
Number of NNZ interaction matrices (K^{max})	117	244	370	551
Number of NNZ matrices between FMM leaves	60	64	49	37
Number of time steps (T)	2033	4345	6647	9957
Size of the simulation box	3.3	7.3	11	16
F_{max}	348	337	335	334
Incomplete FMM coefficient $l = h - 1$	16	18	13	10
Incomplete FMM coefficient $l = 2$	16	36	52	80

Table 5.1: Cone-sphere test cases specifications.



(a) Elapsed time to perform time shifting of V vectors of different length L_e in double precision

(b) Ratio ($T/V.L_e$) to perform time shifting of V vectors of different length L_e in double precision

(c) Elapsed time to perform time shifting of V vectors of different length L_e in single precision

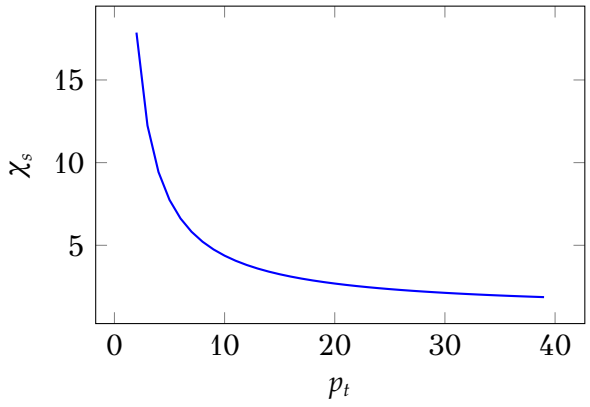
(d) Ratio ($T/V.L_e$) to perform time shifting of V vectors of different length L_e in single precision

Figure 5.12: Time shifting duration on a Intel Xeon $E5 - 2680$ at $2,50GHz$ and with Intel MKL 11.2 (2015.3.187)

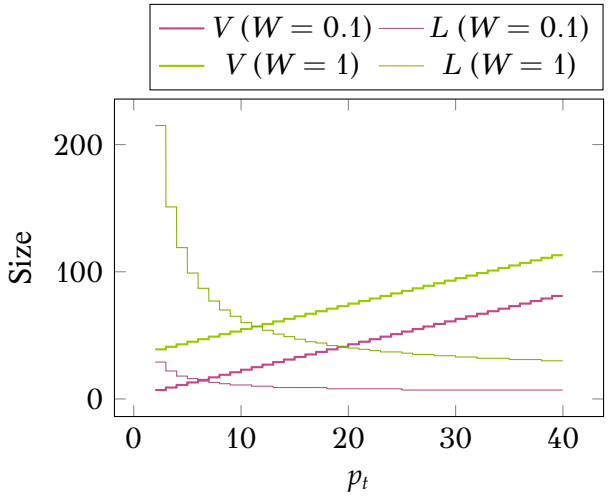
Experimental setup. For the rest of the TD-FMM study we use the following configuration: the nodes are composed by 2 Dodeca-core Haswell Intel® Xeon® E5-2680 at $2,50GHz$ and $128GB$ (DDR4) of shared memory, and we use Gcc 4.9.2, Openmpi 1.8.4 and MKL 11.2 (2015.3.187).

5.5.5 Time-Domain vs. Frequency-Domain Operators

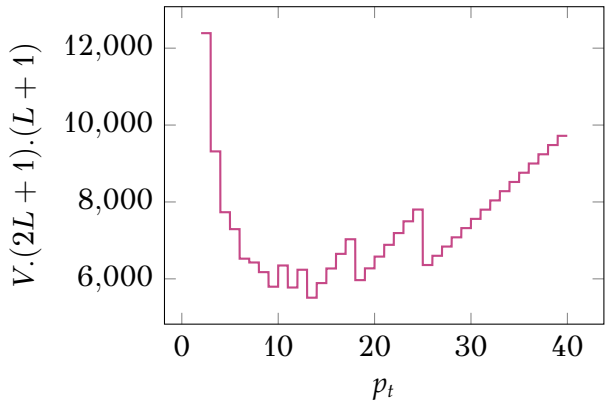
We introduce our FMM operators in the time-domain and the frequency-domain in Section 5.1. In Table 5.2, we compare the solve of the FMM for three configurations: all the operators are computed in the time-domain (Time-domain operators (TD M2L)), all the operators are computed in the frequency-domain (Frequency-domain operators), and an hybrid method where all the operators are in the time-domain except the $M2L$ which is in the frequency-domain (Time-domain operators (FD M2L)). This last method involves Fourier transforms before and after each $M2L$ operation. We see that the time-domain schemes are much faster than the full frequency-domain approach. This is not surprising from the result of the time shift cost in previous Section 5.5.2



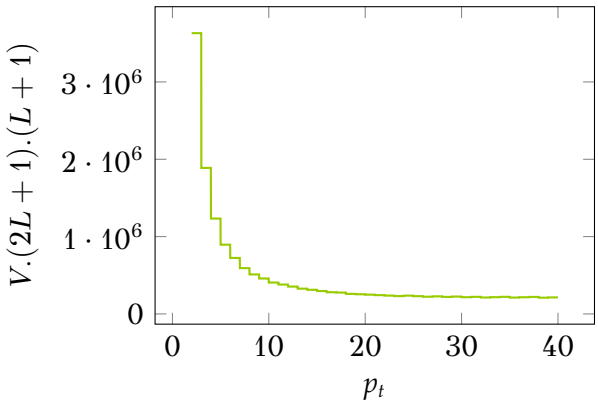
(a) χ_s and p_t to ensure 10^{-2} Accuracy



(b) Number of points in the unit sphere for two box widths ($W = 0.1$ and $W = 1$)



(c) Number of values in a leaf unit sphere after a $P2M$ ($W = 0.1$)



(d) Number of values in a leaf unit sphere after a $P2M$ ($W = 1$)

Figure 5.13: Study of the relation between χ_s and p_t for an accuracy of $\varepsilon_t = 10^{-2}$, $C_\varepsilon = 7$ and two box widths ($W = 0.1$ and $W = 1$). The physical parameters are $f_{max} = 3.482218 \cdot 10^2$ $\Delta t = 1.435867 \cdot 10^{-4}$ and $c = 3.4 \cdot 10^2$.

which increases the complexity, as discussed in Section 5.3.1. The pure frequency-domain approach is not competitive, and we concentrate on the time-domain operators for the rest of the study with possibly the $M2L$ in the frequency-domain. The computation is faster when the $M2L$ is computed in the frequency-domain, and this is the expecting behavior because the convolution product (time-domain) has a larger complexity compared to the term-by-term multiplications equivalent (frequency-domain).

	Time-domain operators (TD M2L)	Time-domain operators (FD M2L)	Frequency-domain operators
Solve Time	58 122 s	53 241 s	97 861s

Table 5.2: Execution time TD-FMM with TD Vs. FD operators to solve the Case C-927 (do not include the construction of the matrices). The solves are done sequentially in double precision.

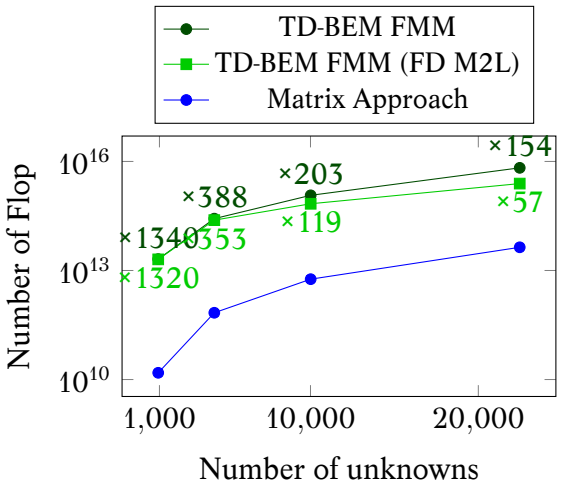
5.5.6 Matrix Approach vs. FMM Approach

The objective of our TD-FMM is to reduce the total simulation time compared to the matrix approach from Chapter 3. The total time includes the construction of the interaction matrices (ob-

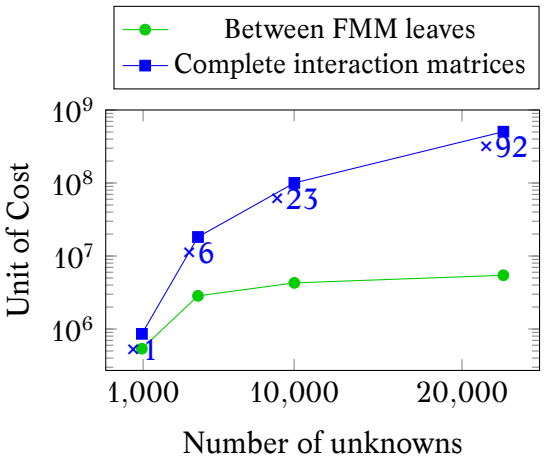
tained from an external black-box) because the FMM computes the interaction matrices between the leaves only. In the rest of the study, we use the FMM operator in the time-domain but also the $M2L$ in the frequency-domain.

Flop/Cost Estimations

Figure 5.14a shows the estimation of the Flop cost to compute the summation stage using the FMM and the matrix approach. For the matrix approach, this number is obtained by multiplying the number of NNZ in the matrices, the number of iterations, the number of right-hand sides and 2 for the plus-equal operation. For the FMM, we develop a kernel estimator which counts the operations that the real kernel would have done. We see that the number of operations in the FMM is drastically much higher but the gap reduces as the problems become larger. Again, computing the $M2L$ in the time-domain reduces the Flop cost. In Figure 5.14b, we provide the complexity to compute the interaction matrices for both approaches; we look at the complexity to compute the interaction matrices between the leaves or for the entire system. Generating the interaction matrices between the leaves is of course much cheaper but it is also more competitive as the problems become larger too. The balance between the costs of the construction and the solve will make the FMM more or less competitive.



(a) Number of Flop for the summation stage using the FMM or the direct approach

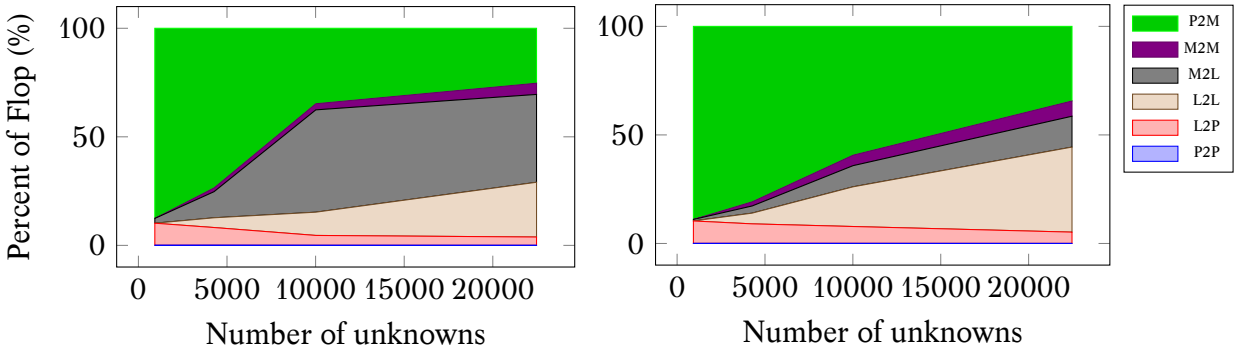


(b) Cost estimation to generate the interaction matrices between the leaves of the FMM ($(N/N_{leaves})^2 \times 11 \times N_{leaves}$) or integrally for the matrix approach (N^2).

Figure 5.14: Flop in the TD-BEM FMM against the matrix approach for the solve stage and cost-estimation to generate the interaction matrices for both cases. The numbers above the slower plot represent the slow-down factors against the faster method.

In Figure 5.15, we provide the details of the cost for the different operators. We compare the two methods with the $M2L$ in the time or in the frequency domain, and we see that from the estimation the $M2L$ in the time-domain is more and more expensive as the problem size increases. The estimations also shows that the $P2M$ is the dominant operator but the cost of the $L2L$ increases with the size of the problems. The $P2M$ is done at each time steps (to propagate the present to the future) and our estimation is based on our implementation where we do not compute the real APS

function, which is based on expensive mathematical operators, but we use a linear interpolation. The $L2L$ is more and more important because as the problems become larger, the signals cover more time steps especially from the upper $M2L$.



(a) Time-domain $M2L$ **(b)** Frequency-domain $M2L$

Figure 5.15: Relative cost for the different FMM TD operators in terms of Flop.

We also test to reduce the height of the tree, and the results are not shown in the current study because it always increases the overall cost in our cases. For a given configuration, it is clear that reducing the size of the tree increases directly the cost of the matrix construction. But it also increases the cost of the computation stage, and for example the $C - 22468$ case has a cost of $4.1 \cdot 10^{15} Flop$ for $h = 5$, whereas from the table it has a cost of $2.5 \cdot 10^{15} Flop$ for $h = 6s$.

Sequential Comparison

In Table 5.3 we compare the FMM approach with operators in the time-domain against the matrix approach. The test case C-927 has a low number of unknowns, but it is already expensive because it has 2033 time iterations. We see that the total execution time is not negligible especially for the FMM. The FMM is clearly slower than the direct matrix approach for this test case but as expected, the FMM is faster for the construction of the matrices. Therefore, the competitiveness of the FMM is tied to these ratio between the construction of the matrices and the solve costs. The solve stage is proportional to the number of right-hand sides whereas the construction of the matrices is unchanged (because the same matrices are used) and this is an advantage for the matrix approach. In order to study larger test cases, we have to use a parallel implementation even if it certainly includes efficiency difference from the parallel implementations rather than the underlying methods. In addition, we see from the Flop estimations, Figure 5.15, and the results of the test case C-927, Figure 5.3, it appears better to compute the $M2L$ in the frequency-domain.

Stages	TD-FMM	TD-FMM (FD M2L)	Matrix approach
Construction	76 s	76 s	242 s
Solve	58 122 s	53 241 s	7.8 s
Total	58 198 s	53 317 s	249.8 s

Table 5.3: Execution time TD-FMM with TD operators Vs. matrix approach to solve the Case C-927. The solves are done in sequential.

Shared Memory Parallel Comparison

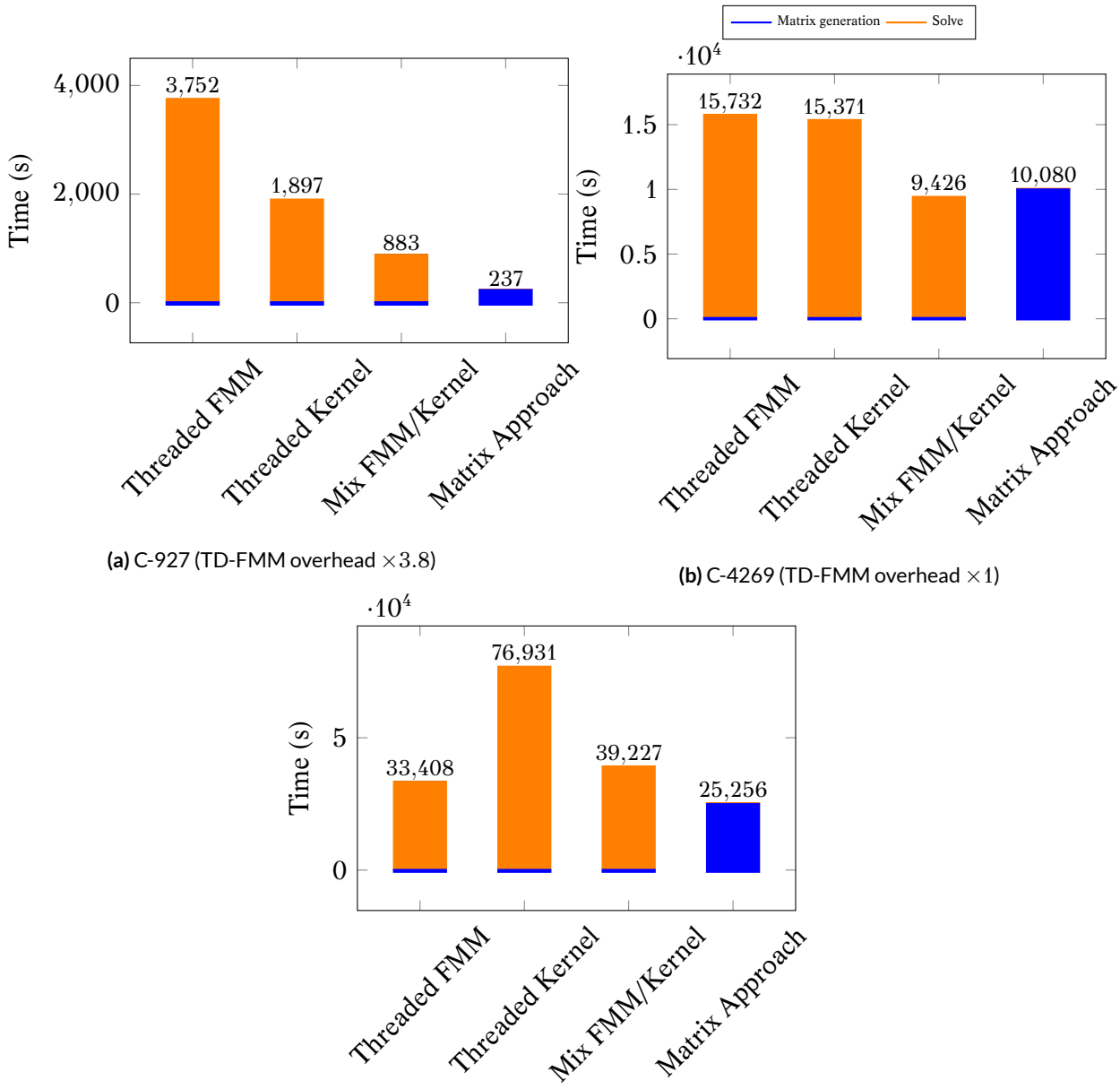
A main difference between our TD-FMM kernel and the usual particle-based FMM kernels is the high cost of the operators and the small tree height. The tree height is reduced not only because the far-field is expensive but also because the size of the discretization mesh elements restrained the width of the leaves. ScalFMM proposes several parallelization strategies on shared memory as discussed in Chapter 3. From the parallel efficiency results, see Section 4.7, the *parallel-for* is efficient to compute particle-based simulations. However, the small number of cells/leaves restricts the parallelism; for example the C-927 case has only 16 leaves. One solution to introduce more parallelism is to parallelize not only the FMM algorithm but also the operators.

In Figure 5.16, we present the time to compute the different study cases using the FMM or the direct matrix approach. For the FMM, we study three different parallelization schemes; the parallelization is done at the FMM level (Threaded FMM), the parallelization is done inside the kernel (Threaded kernel), or the parallelization is done at both levels (Mix FMM/Kernel). We observe that for the small case C-927, it is better to use inner parallelism directly in the kernel. This is because there are only 16 leaves for 24 threads and moreover the leaves are partitioned into 27 colors in the $P2P$. But, as the size of the problems become larger, there is enough parallelism and it is better to parallelize at the FMM level. The best parallelization scheme mixes both parallelism, it uses 4 threads at the FMM level where each of them creates 6 threads inside the operators. This solutions has good memory properties by having only 6 threads which work on the same data/cells but only 4 threads at the FMM layer. However, when we compare the TD-BEM to the direct matrix approach, even if the gap seems to reduce the speedup is stuck, and it is not competitive for the number of unknowns we study.

In Figure 5.17, we provide the percentage in terms of simulation time for the different FMM operators. It matches our Flop cost estimation from Figure 5.14a.

Memory Occupancy Evaluation

In Figure 5.18, we give the memory occupancy for the matrix approach and the TD-FMM. In addition, we show the size of the complete interaction matrices which is the number of non zero times the size of the floating point data type. The matrix approach has a limited overhead against the size of the interaction matrices because it only allocates the a^n and l^n vectors. On the other hand, the TD-FMM requires much more memory even if the factor decreases against the matrix approach as the problems become larger. Moreover, the size of the FMM is proportional to the number of right-hand sides because we need one octree per right-hand side since we keep information inside the cells between the iterations. Whereas, in the matrix approach we use the same matrices, and we see from the figure that the memory cost does not increase a lot as we increase the number of right-hand side.



(a) C-927 (TD-FMM overhead $\times 3.8$)

(b) C-4269 (TD-FMM overhead $\times 1$)

(c) C-10012 (TD-FMM overhead $\times 1.4$)

Figure 5.16: Comparison of the TD-FMM, with operators in the time-domain and the $M2L$ in the frequency-domain, against the matrix approach on one node using 24 threads. In the *Threaded FMM* the parallelism is done above the tree level by ScalFMM. For the *Threaded-Kernel* the parallelism is done inside the kernel above the Gauss points of the unit spheres. The *Mix FMM/Kernel* is based on nested parallelization, 4 threads works on the tree levels and each of them creates 6 threads inside the kernel. The captions of the different cases show the overhead of the FMM TD-BEM against the matrix approach.

Hybrid MPI/OpenMP Comparison

In Figure 5.19, we show the execution times for the TD-FMM and the matrix approach using 2 nodes. The matrix approach has an efficient parallelization strategy which is difficult to bypass. Moreover, the construction of the matrices has an superlinear speedup, which is difficult to study because it is an external module. The system is divided between the two nodes, and therefore the parallelism at the FMM level is efficient for the C-10012 case because in C-4269 there are not enough leaves/cells. Finally, the gap remains almost the same compared to the shared memory

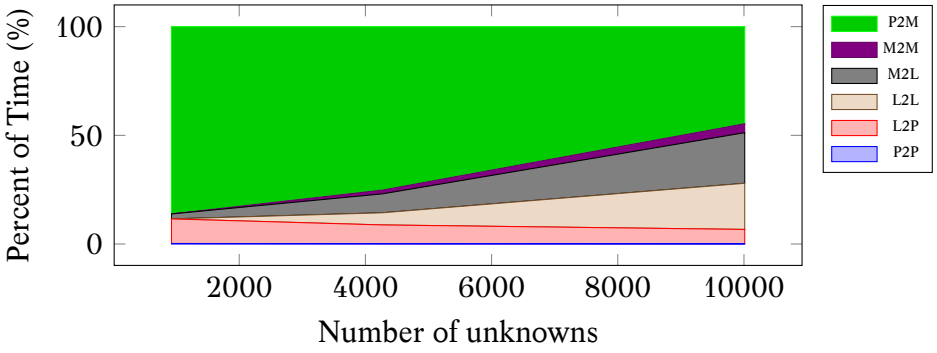


Figure 5.17: Relative cost for the different FMM TD operators from the executions using 24 threads.

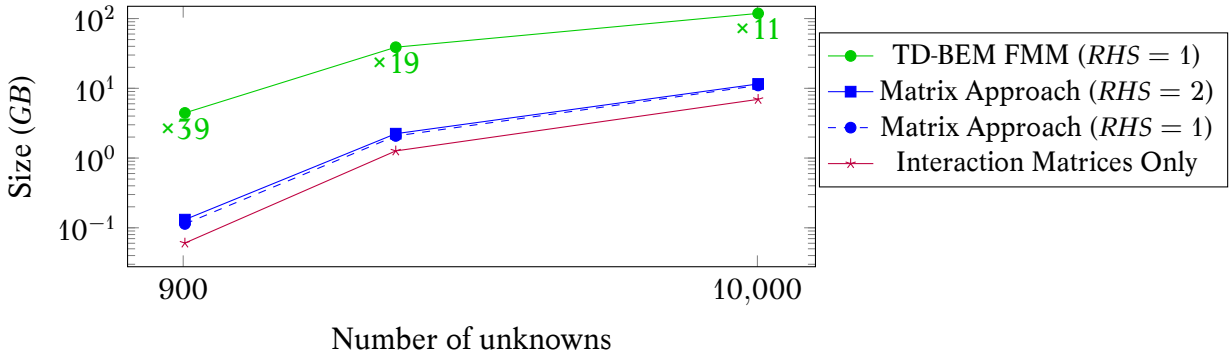


Figure 5.18: Memory occupancy for the TD-BEM FMM against the matrix approach for the cone-sphere test cases in double precision. The numbers above the *TD-BEM FMM* plot represent the overhead against the *Matrix Approach* method.

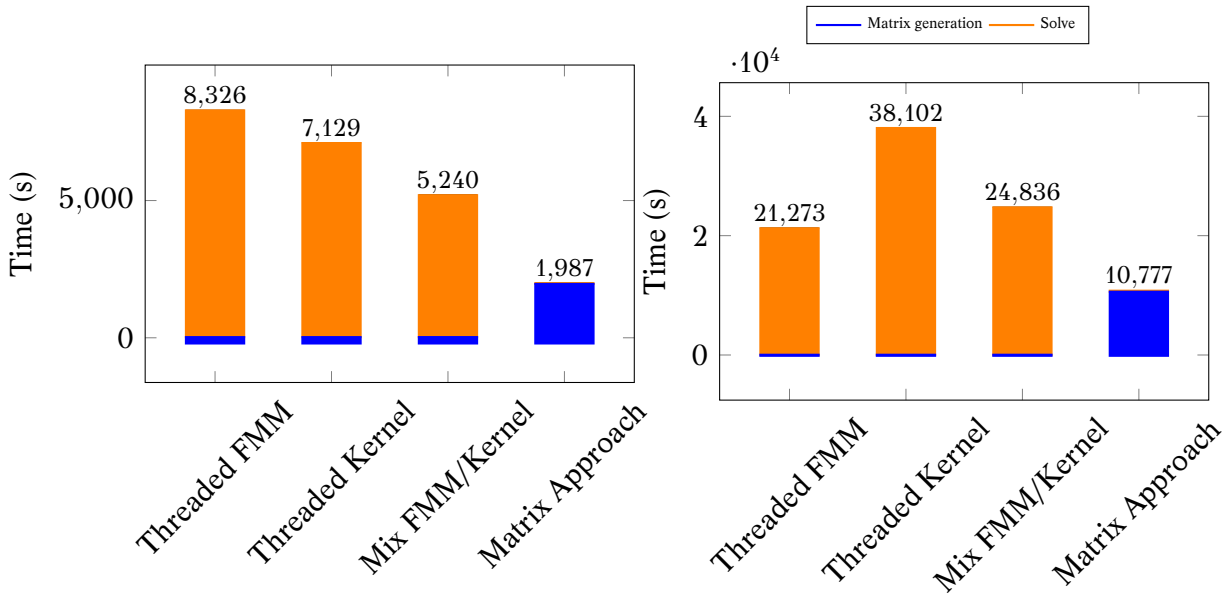
executions, and our FMM is slower.

5.6 Summary and Perspective

Our preliminary study is empirical, and our comparison is more on our implementations rather than on the methods. The preliminary results illustrate how difficult it is to bypass the matrix approach and show that our implementation of the TD-BEM with the FMM is clearly not competitive. The construction of the matrices is the dominant part of the matrix approach solver, but we do not have the hand on this layer. However, any optimizations and improvements of this part will make the matrix approach even better against the FMM based solver. Moreover, it seems that the generator performs many hard drive accesses, and implementing a pure in-core version might lead to a huge speed-up.

Increasing the number of right-hand sides is also a drawback to the FMM approach. From a memory standpoint, we need one octree per right-hand side whereas the matrix approach only allocates extra vectors. From the computational side, the matrix construction cost is independent of the number of right-hand sides, whereas the computation is proportional to it, which is also an advantage in the matrix approach.

It does not seem interesting to compute the TD-FMM using upward/downward operators in the frequency-domain. Therefore, the future work on the method should focus on the time-domain operators with the $M2L$ in the frequency-domain, and with the support of low level optimizations;



(a) C-4269 (TD-FMM overhead $\times 2.7$)

(b) C-10012 (TD-FMM overhead $\times 2$)

Figure 5.19: Comparison of the TD-FMM, with operators in the time-domain and the $M2L$ in the frequency-domain, against the matrix approach on one node using 2 nodes and 24 threads on each. In the *Threaded FMM* the parallelism is done above the tree level by ScalFMM. For the *Threaded-Kernel* the parallelism is done inside the kernel above the Gauss points of the unit spheres. The *Mix FMM/Kernel* is based on nested parallelization, 4 threads works on the tree levels and each of them creates 6 threads inside the kernel. The captions of the different cases show the overhead of the FMM TD-BEM against the matrix approach.

for example, the use of explicit SIMD intrinsics might lead to a great speedup (up-to 8 times faster). In addition, it would be very useful to perform a theoretical study to evaluate the cost and real complexity of the method even by restraining it to few test cases. It will tell us when it is advantageous to use the FMM solver and from which number of unknowns. In terms of numerical optimization, we do not play with different accuracies, and especially we do not look at the behavior if we decrease the accuracy at the upper levels.

Conclusion

This thesis studies the development of an efficient solver for the time-domain BEM for the wave equation on modern HPC architectures.

The original formulation is based on the SpMV and we explain how the performances of this operator can be improved by permuting the matrices and creating dense blocks around the NNZ values. We define the *max-block-score* heuristic which has a low cost and gives good permutations without a $2OPT$ refining. In addition, we present advanced SpMV kernels for unaligned blocks on CPU and a custom storage on GPU both with aggressive compile-time optimizations. Nevertheless, the zero padding inside the blocks remains huge (up to 50% on CPU) and the effective Flop-rate of our kernels decreases by the same proportion. Besides, we deal with a set of matrices and finding a single ordering for all of them to avoid the permutation of the vectors appears extremely complex. Therefore, it appears clear that a SpMV-based solver is not appropriate especially to target accelerators.

To improve the Flop/memory ratio, we propose a different computational order. We permute the loops of the summation stage and obtain the slice matrices that have one dense vector per row. We show that by working with n_g summation vectors together we have to load $d + n_g + d \times n_g$ data to perform $n_g \times 2d$ Flop (with d the length of the dense vector on the row) in the so-called multi-vectors/vector product. Naive implementations of this operator do not have good performances. On the CPU, it is mandatory to use SIMD operands explicitly even if it leads to a more complex algorithm. Moreover, we provide a kernel in assembly and show that it is pertinent to increase the registers usage. The SIMT and hardware specificities of the GPU make the data blocking and loop unrolling imperative. We present two methods to extract blocks from the slice matrices, but only the Contiguous-Blocking can be used in realistic simulations. In fact, this method generates blocks with few extra-zero padding, it is easy to parametrize and it achieves a high Flop-rate even for small block widths. Based on these kernels, our matrix approach solver is parallelized using a hybrid MPI/OpenMP strategies over homogeneous or heterogeneous nodes. A new balancing heuristic successfully balances the summation inside a heterogeneous node, and our application has a good parallel efficiency up to 30 homogeneous nodes and 8 heterogeneous nodes with 4 GPUs each. However, the call to the linear solver for the matrix M^0 becomes the bottleneck as the number of node increases. In fact, we have a small number of unknowns, an important number of nodes, and we solve the same system several times which is not a classic usage of the state-of-the-art linear solvers. Moreover, the generation of the matrices is finally more expensive than the solve/computation, and both stages have a quadratic complexity. Therefore, our study looks at the potential benefit of the FMM in our TD-BEM.

To create our FMM based solver, we work on the FMM algorithm as a generic technique, and

we describe various parallelization strategies. Starting from a sequential FMM, we incorporate the *parallel-for* work division and improve the expression of the parallelism until a pure task-based FMM over a runtime system. This *tasks-and-dependencies* approach comes with two mains problematics related to the expression of the dependencies and the data structure. We show that the FMM is naturally described using commutative operations even if this propriety is not widespread among the runtime systems. We present the group-tree which is a tree data structure made for the FMM over a runtime system; it relies on plain-old data that can be moved easily between memories, it separates the symbolic/multipole/local parts, and it allows to parametrize the blocking granularity. We extend our shared memory implementations with two distributed memory parallelization strategies. The first relies on a classic MPI/OpenMP development with a communication hiding system. The second extends the *tasks-and-dependencies* for distributed memory, and we explain how this is possible once there is a system to manage the calls to the MPI functions asynchronously. The performance results are diverse. The *parallel-for* and the *tasks-and-dependencies* methods on shared memory, and the hybrid MPI/OpenMP and StarPU-MPI on distributed memory are all competitive. However, for small test cases the runtime overhead makes the OpenMP approaches more appropriate.

We attempt to reduce the complexity of the matrix approach using the FMM in the summation stage. The underlying kernel is complex and asks for various choices in its implementation and its parametrization. From our results, the p_t parameter must be carefully chosen otherwise the complexity might increase dramatically. The time-shift operation is widely used and is critical for the accuracy and the performance. This is one of the reasons that make the frequency-domain operators much slower than the time-domain ones. As expected, using the FMM reduces the cost of the matrix generation which is the dominant stage of the matrix approach. However, our FMM implementation is 1.4 times slower than the matrix approach such that the total time is not improved with the FMM up-to 10 012 unknowns. The classical strategies are not sufficient to parallelize this kernel, and we need two levels of parallelism; one in the FMM and one in the operators.

From this preliminarily results, we need a theoretical study to know if the FMM method will lead to better performance for larger problems. It is a difficult question, and to provide the first answer a complexity study should be done on a defined test case. In the implementation side, our FMM kernel is not highly optimized and using SIMD instructions or a different interpolation scheme could potentially reduce the execution time. Finally, we do not assess in details our TD-FMM on distributed memory, but it will certainly open new questions regarding the balancing strategies between the nodes and inside the nodes at the top of the tree.

Perspectives

The results of our matrix approach point-out that, when the number of node increases, the linear solver becomes a critical component. Therefore, we have to find the best existing solver or to develop a new one that matches the problem specificities: the small dimension of the matrix, a large number of nodes and up-to thousands of solves. Iterative solvers are certainly well adapted, but the gain will also depend on the underlying parallelization. In fact, we show that our configuration is on the limit of the sparse solvers and that increasing the number of nodes does not decrease the solve time. One possible improvement is to reduce the number of communications by computing the same solve in several subgroups of processes. For a given problem, if a solver gives its best performance with m processes, we create np/m subgroups of processes. Therefore, there will be one global reduction after the summation stage, but then only the processes inside a subgroup will communicate during the solve and for the broadcast of the result.

We should assess the robustness of our FMM hybrid parallelization algorithms using a large number of cores/nodes. In addition, we now have a stable StarPU-based implementation for distributed homogeneous architectures, but the main advantage of using StarPU is to incorporate accelerators easily. Therefore, a future work will be to implement the $P2P$ and the $M2L$ for our fastest approximation kernel in OpenCL to target both Inel Xeon Phis and NVidia GPUs. In a different direction, it would be interesting to implement an adaptive FMM. In fact, some problems can be computed faster if the depth of the tree is variable and directly mapped on the particles/mesh density. However, the parallelization of an adaptive FMM with the *tasks-and-dependencies* scheme will be challenging and potentially require a new tree data-structure to reduce the number of dependencies.

Our BEM FMM solver can be optimized at several levels. From our results, the best configuration is obtained by mixing the TD operators with the FD $M2L$. Therefore, the next developments should focus on these operators and to optimize them using vectorization and an appropriate data structure. Applying the vectorization over on the vectors in the time direction might not give efficient results. On the other hand, we take advantage of the fact that the same operation is applied to all the functions defined over the unit sphere. Therefore, the best pattern will be to block the vectors in the sphere discretization direction by the size of the SIMD data-type. At the algorithm level, we always move downward the results of the $M2L$: in our incomplete FMM algorithm, when we compute the $M2L$ at level l, we also compute the $L2L$ from level l to $h-2$. Nevertheless, we can reduce the computational cost and the memory footprint by transferring only a part of the data. Finally, the memory footprint of the $M2L$ transfer matrices increases with the size of the problem such that it might become a critical issue. We can imagine finding a relation between the different matrices in order to reduce the memory occupancy even if it implies a computational extra cost.

For example, if there is a symmetry or a rotation property, we may succeed to have fewer matrices and to generate the appropriate values on the fly.

Appendices

A

Acoustics TD-BEM Formulation

This section introduces the mathematical details of our problem formulation. The given problem description and formulation have been taken from [3] and the TD formulation was originally introduced in the context of electromagnetism by [2].

A.1 Problem Formulation

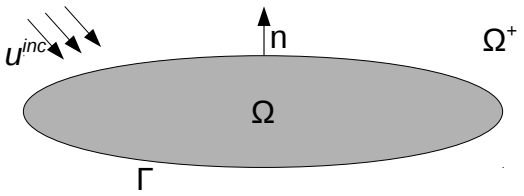


Figure A.1: Problem domains

The variable u_{inc} is an incident wave given by the problem definition and the aim is to find the reflection of the wave over the object Ω of boundary Γ. We consider the problem in a fluid and we name c the wave velocity in the fluid and $\vec{n}$ the unit norm in the exterior of the boundary. In the exterior domain Ω^+, the problem P^+ is to find $u^+(x,t)$ solution of

$$(P^+)\begin{cases} \Delta u^+ - \frac{1}{c^2}\frac{\partial^2 u^+}{\partial t^2} = 0 & \text{in } \Omega^+ \times \mathbb{R}^+, \\ \frac{\partial u^+}{\partial n} - \frac{1}{c\eta}\frac{\partial u^+}{\partial t} = -(\frac{\partial u_{inc}}{\partial n} - \frac{1}{c\eta}\frac{\partial u_{inc}}{\partial t}) & \text{in } \Gamma \times \mathbb{R}^+, \\ u^+(x,0) = 0 & \text{in } \Omega^+, \\ \frac{\partial u^+}{\partial t}(x,0) = 0 & \text{in } \Omega^+. \end{cases} \tag{A.1}$$

We associate to P^+ a problem P^- in the interior domain of Ω called Ω^-: find $u^-(x,t)$ such as

$$(P^-)\begin{cases} \Delta u^- - \frac{1}{c^2}\frac{\partial^2 u^-}{\partial t^2} = 0 & \text{in } \Omega^- \times \mathbb{R}^+, \\ \frac{\partial u^-}{\partial n} - \frac{1}{c\eta}\frac{\partial u^-}{\partial t} = -(\frac{\partial u_{inc}}{\partial n} - \frac{1}{c\eta}\frac{\partial u_{inc}}{\partial t}) & \text{in } \Gamma \times \mathbb{R}^+, \\ u^-(x,0) = 0 & \text{in } \Omega^-, \\ \frac{\partial u^-}{\partial t}(x,0) = 0 & \text{in } \Omega^-. \end{cases} \tag{A.2}$$

We denote by Φ and p the tangential jumps of u and $\partial u/\partial n$ on Γ defined by

$$\begin{cases} \Phi(x) = u^-_{|\Gamma}(x,t) - u^+_{|\Gamma}(x,t) & x \in \Gamma, t \geq 0, \\ p(x) = \left(\frac{\partial u^-}{\partial n}\right)_{|\Gamma}(x,t) - \left(\frac{\partial u^+}{\partial n}\right)_{|\Gamma}(x,t) & x \in \Gamma, t \geq 0. \end{cases} \tag{A.3}$$

Rather than solving the problem on the whole space, we write it on the surface Γ. Let introduce the following surfacic operators

$$\begin{cases} Sp(x,t) = \frac{1}{4\pi}\int_\Gamma \frac{p(y,t-|x-y|/c)}{|x-y|}dy, \\ Kp(x,t) = \frac{1}{4\pi}\int_\Gamma \frac{\partial}{\partial n_x}\frac{p(y,t-|x-y|/c)}{|x-y|}dy, \\ K'\Phi(x,t) = \frac{1}{4\pi}\int_\Gamma \frac{\partial}{\partial n_y}\frac{\Phi(y,t-|x-y|/c)}{|x-y|}dy, \\ D\Phi(x,t) = \frac{1}{4\pi}\oint_\Gamma \frac{\partial^2}{\partial n_x \partial n_y}\frac{\Phi(y,t-|x-y|/c)}{|x-y|}dy. \end{cases} \tag{A.4}$$

Then, p and Φ are solutions in $\Gamma \times \mathbb{R}^+$ of

$$\begin{cases} (Kp - D\Phi) + \frac{1}{2c\eta}\frac{\partial \Phi}{\partial t} = -\frac{\partial u_{inc}}{\partial n}, \\ \frac{\partial}{\partial t}(Sp - K'\Phi) + \frac{c\eta}{2}p = -\frac{\partial u_{inc}}{\partial t}. \end{cases} \tag{A.5}$$

Let introduce a new variable

$$P(x,t) = \begin{cases} c\int_0^t p(x,\tau)d\tau & t \geq 0, \\ 0 & t < 0. \end{cases} \tag{A.6}$$

We replace p by P in Equation A.5, and multiply the first equation by $-c$ to obtain our formulation problem

$$\begin{cases} \frac{\partial}{\partial t}(S\frac{\partial_t P}{c} - K'\Phi) + \frac{\eta}{2}\partial_t P = -\frac{\partial u_{inc}}{\partial t}, \\ cD\Phi - K\frac{\partial P}{\partial t} - \frac{1}{2\eta}\frac{\partial \Phi}{\partial t} = c\frac{\partial u_{inc}}{\partial n}. \end{cases} \tag{A.7}$$

A.2 Weak Formulation

The weak formulation of Equations A.7 are classically obtained by multiplying by $\partial\Psi'/\partial t$ for the first one to obtain Equation A.8, and the second by q which lead to Equation A.9.

$$\begin{aligned}
&-\int_{\mathbb{R}}\int_{\Gamma\times\Gamma}\frac{\partial}{\partial n_x}\frac{\partial_t P(y,t-|x-y|/c)}{4\pi|x-y|}\frac{\partial\Psi'}{\partial t}(x,t)dxdydt\\
&-\frac{1}{c}\int_{\mathbb{R}}\int_{\Gamma\times\Gamma}\frac{\vec{n}(x).\vec{n}(y)}{4\pi|x-y|}\frac{\partial^2\Phi}{\partial t^2}(y,t-\frac{|x-y|}{c})\frac{\partial\Psi'}{\partial t}(x,t)dxdydt\\
&-c\int_{\mathbb{R}}\int_{\Gamma\times\Gamma}\frac{1}{4\pi|x-y|}\vec{rot}_\Gamma\Phi(y,t-\frac{|x-y|}{c})\vec{rot}_\Gamma\frac{\partial\Psi'}{\partial t}(x,t)dxdydt\\
&-\frac{1}{2\eta}\int_{\mathbb{R}}\int_{\Gamma}\frac{\partial\Phi}{\partial t}(x,t)\frac{\partial\Psi'}{\partial t}(x,t)dxdt\\
&=\int_{\mathbb{R}}\int_{\Gamma}c\frac{\partial u_{inc}}{\partial n}(x,t)\frac{\partial\Psi'}{\partial t}(x,t)dxdt.
\end{aligned}\tag{A.8}$$

$$\begin{aligned}
&\frac{1}{c}\int_{\mathbb{R}}\int_{\Gamma\times\Gamma}\frac{1}{4\pi|x-y|}\frac{\partial^2 P}{\partial t^2}(y,t-|x-y|/c)q(x,t)dxdydt\\
&-\int_{\mathbb{R}}\int_{\Gamma\times\Gamma}\frac{\partial}{\partial t}\frac{\partial}{\partial n_y}\frac{\Phi(y,t-|x-y|/c)}{4\pi|x-y|}q(x,t)dxdydt\\
&+\frac{1}{2}\int_{\mathbb{R}}\int_{\Gamma}\partial_t P(x,t)q(x,t)dxdt\\
&=\int_{\mathbb{R}}\int_{\Gamma}-\frac{\partial u_{inc}}{\partial t}(x,t)q(x,t)dxdt.
\end{aligned}\tag{A.9}$$

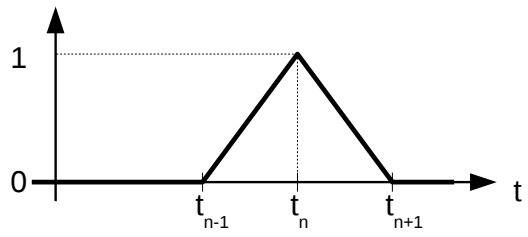
These two equations gives the right expression of the different operators involved in the problem.

A.3 Discretizations

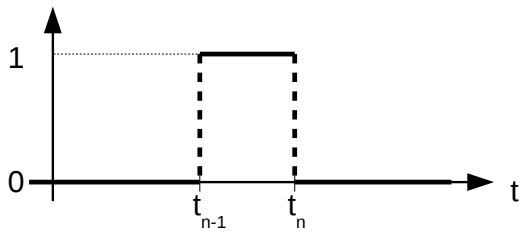
Time Discretization. The time interval $[0,T]$ is discretized by a uniform time step Δt. The time points are noted by $t_n = n\Delta t$ and we consider a linear approximation in time. The basis functions are written $\gamma^n(t)$ ($n \geq 1$) and illustrated by Figure A.2a. We also introduce the function basis $\chi_n(t)$ ($n \geq 0$) presented in Figure A.2b. Then we obtain the following relation for the time derivatives of the basis functions

$$\begin{cases}\frac{\partial\gamma_n}{\partial t} = \frac{1}{\Delta t}[\chi_n - \chi_{n+1}]\\ \frac{\partial^2\gamma_n}{\partial t^2} = \frac{1}{\Delta t}[\delta_{t_{n+1}} - 2\delta_{t_n} + \delta_{t_{n-1}}].\end{cases}\tag{A.10}$$

Space Discretization. The problem is discretized in space by a $P1$ finite element method (Figure A.3b). The surface Γ of the object is approximated by a triangulation T_h composed by N_T triangular elements and N_S vertices. For the function $\Phi(x,t)$, which represents the pressure jump, we use a discretization by a linear polynomial in space. Each basis function φ_i is associated to a



(a) Shape of the function γ_n



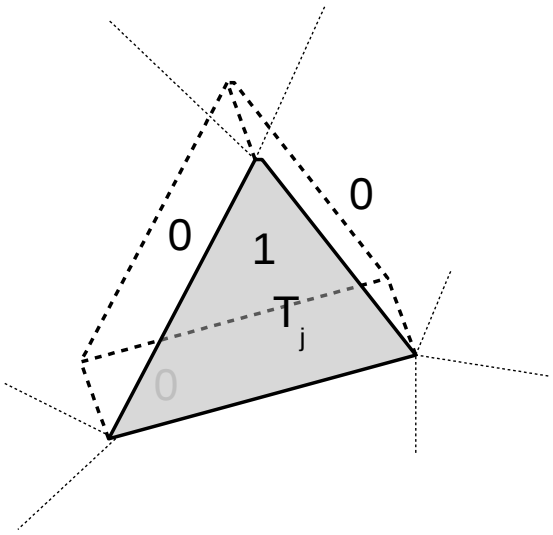
(b) Shape of the function χ_n

Figure A.2: Basis functions.

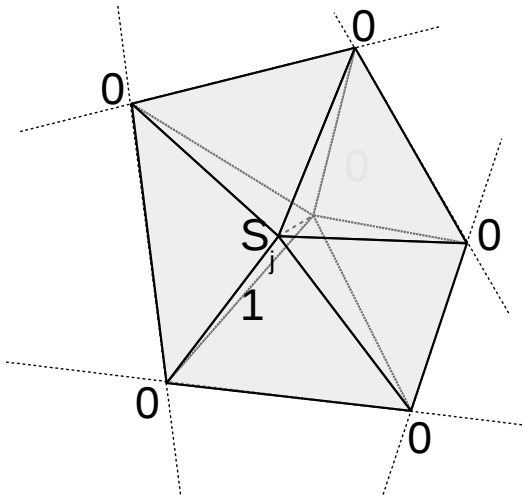
vertex S_i for $1 \leq i \leq N_S$ is defined by

$$\begin{cases} \varphi_i(S_j) = \delta_{ij}, \\ \varphi_i \text{ is } P1 \text{ on each triangle.} \end{cases} \tag{A.11}$$

Therefore, in a triangle, $\varphi_j(M)$ is the barycentric coordinate of M on the vertex j. For the function $p(x,t)$, which represents the normal derivative of the pressure, we consider a constant approximation ($P0$) on each triangle. The basis functions are p_j with $1 \leq j \leq N_T$. The characteristic function of the triangle T_j takes the value 1 on the triangle T_j and 0 on the others as presented in Figure A.3a.



(a) $P0$ discretization



(b) $P1$ discretization

Figure A.3: Discretizations.

A.4 Notation

The functions Φ and P are respective solutions of Equation A.8 and Equations A.9 and are approximated by the following expansions

$$\begin{cases} \Phi_h(x,t) = \sum_{1 \leq j \leq N_S} \sum_{m \geq 1} a_j^m \gamma_m(t) \varphi_j(x), \\ P_h(x,t) = \sum_{1 \leq j \leq N_T} \sum_{m \geq 1} b_j^m \gamma_m(t) p_j(x). \end{cases} \tag{A.12}$$

We denote by A^m and B^m, the vectors of Φ and P respectively, a time t_m

$$\left\{ A^m = [a_j^m]_{1\leq j\leq N_S} \text{ and } B^m = [b_j^m]_{1\leq j\leq N_T}. \right. \tag{A.13}$$

Finally, we consider the test functions Ψ and q given by

$$\begin{cases} \dfrac{\partial \Psi}{\partial t}(x,t) = \chi_n(t)\varphi_i(x) \\ q(x,t) = \chi_n(t)p_i(x). \end{cases} \tag{A.14}$$

A.5 Rigid Object Discretization

If we first focus on a rigid object Ω (constant in time), the term P and η are removed and the system becomes

$$\begin{aligned} &-\frac{1}{c}\int_{\mathbb{R}}\int_{\Gamma\times\Gamma}\frac{\vec{n}(x).\vec{n}(y)}{4\pi|x-y|}\frac{\partial^2\Phi}{\partial t^2}(y,t-\frac{|x-y|}{c})\frac{\partial\Psi}{\partial t}(x,t)dxdydt \\ &-c\int_{\mathbb{R}}\int_{\Gamma\times\Gamma}\frac{1}{4\pi|x-y|}\vec{rot}_\Gamma\Phi(y,t-\frac{|x-y|}{c})\vec{rot}_\Gamma\frac{\partial\Psi}{\partial t}(x,t)dxdydt \\ &=c\int_{\mathbb{R}}\int_{\Gamma}\frac{\partial u_{inc}}{\partial n}(x,t)\frac{\partial\Psi}{\partial t}(x,t)dxdt. \end{aligned} \tag{A.15}$$

A.6 Interaction Matrices

From the left hand-side, with basis function $\Phi(x,t) = \varphi_j(x)\gamma_n(t)$ and test function $\partial\Psi/\partial t(x,t) = \varphi_i(x)\chi_m(t)$ we obtain

$$\begin{aligned} &-\frac{1}{c}\int_{\Gamma\times\Gamma}\frac{\vec{n}(x).\vec{n}(y)}{4\pi|x-y|}\varphi_i(y)\varphi_i(x)\left[\int_{\mathbb{R}}\frac{\partial^2\gamma_n}{\partial t^2}(t-\frac{|x-y|}{c})\chi_m(t)dt\right]dxdydt \\ &-c\int_{\Gamma\times\Gamma}\frac{1}{4\pi|x-y|}\vec{rot}_\Gamma\varphi_i(y)\vec{rot}_\Gamma\varphi_j(x)\left[\int_{\mathbb{R}}\gamma_n(t-\frac{|x-y|}{c})\chi_m(t)dt\right]dxdydt. \end{aligned} \tag{A.16}$$

From γ_n and χ_m definitions and performing an integration by part, the first time integral becomes

$$\begin{aligned} &\int_{\mathbb{R}}\frac{\partial^2\gamma_n}{\partial t^2}(t-\frac{|x-y|}{c})\chi_m(t)dt \\ &=\int_{\mathbb{R}}\frac{1}{\Delta t}\left[\delta_{t_{n+1}}-2\delta_{t_n}+\delta_{t_{n-1}}\right](t-\frac{|x-y|}{c})\chi_m(t)dt \\ &=\frac{1}{\Delta t}\left[\chi_m(t_{n+1}+\frac{|x-y|}{c})-2\chi_m(t_n+\frac{|x-y|}{c})+\chi_m(t_{n-1}-\frac{|x-y|}{c})\right] \\ &=\frac{1}{\Delta t}\left[\chi_1(\frac{|x-y|}{c}-t_{m-n-2})-2\chi_1(\frac{|x-y|}{c}-t_{m-n-1})+\chi_1(\frac{|x-y|}{c}-t_{m-n})\right]. \end{aligned} \tag{A.17}$$

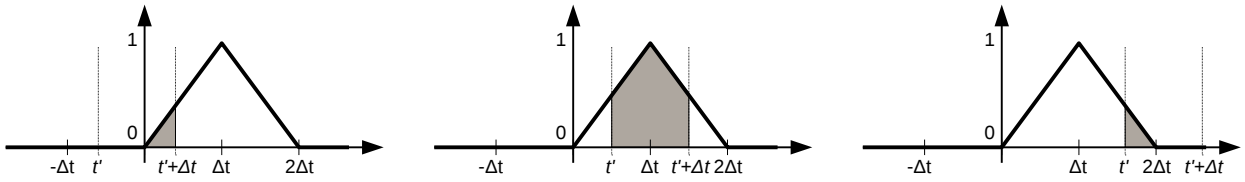
The value of this integral depends only of the difference between indexes m and n, from here we have $k = m - n$. The term $\chi_1(|x-y|/c - t_k)$ reduces the integration on $\Gamma\times\Gamma$ on the values of x

and y such as $|x-y|/c \in [t_k, t_k + \Delta t]$. The second time integration writes

$$\int_{\mathbb{R}} \gamma_n(t - \frac{|x-y|}{c})\chi_m(t)dt = \int_0^{\Delta t} \gamma_1(t + t_k - \frac{|x-y|}{c})dt. \tag{A.18}$$

Since γ_1 is not null between 0 and $2\Delta t$, this integration is not null for all values $t' = t_k - |x-y|/c$ inside $-\Delta t$ and $2\Delta t$ and is equal to

$$\begin{cases} \frac{1}{2\Delta t}(t' + \Delta t)^2 \text{if } t' \in [-\Delta t, 0] \text{ (Figure A.4a)} \\ \frac{1}{2\Delta t}((t' + \Delta t)^2 - 3t'^2) \text{if } t' \in [0, \Delta t] \text{ (Figure A.4b)} \\ \frac{1}{2\Delta t}(2\Delta t - t')^2 \text{if } t' \in [\Delta t, 2\Delta t] \text{ (Figure A.4c).} \end{cases} \tag{A.19}$$



(a) $-\Delta t \leq t' \leq 0$ **(b)** $0 \leq t' \leq \Delta t$ **(c)** $\Delta t \leq t' \leq 2\Delta t$

Figure A.4: Integration of φ_1 between t' and $t' + \Delta t$

Finally, we obtain that the interaction term between the functions $\varphi_j(x)\gamma_n(t)$ and $\varphi_i(x)\chi_m(t)$ depends only on (i,j) in space and $k = m - n$ in time. We call M^k the matrix of dimension $N_S \times N_S$ whose element (i,j) is given by the Equation A.16. More precisely, we have

$$\begin{aligned} M^k_{ij} = -\frac{1}{c\Delta t}\Big(& \\ & \int\int_{t_k \leq \frac{|x-y|}{c} \leq t_{k+1}} \frac{\vec{n}(x).\vec{n}(y)}{4\pi|x-y|}\varphi_i(x)\varphi_j(y)dxdy \\ & - 2\int\int_{t_{k-1} \leq \frac{|x-y|}{c} \leq t_k} \frac{\vec{n}(x).\vec{n}(y)}{4\pi|x-y|}\varphi_i(x)\varphi_j(y)dxdy \\ & + \int\int_{t_{k-2} \leq \frac{|x-y|}{c} \leq t_{k-1}} \frac{\vec{n}(x).\vec{n}(y)}{4\pi|x-y|}\varphi_i(x)\varphi_j(y)dxdy \\ & + \int\int_{t_k \leq \frac{|x-y|}{c} \leq t_{k+1}} \frac{\vec{rot}\varphi_i(x)\vec{rot}\varphi_j(y)}{4\pi|x-y|}\frac{c^2}{2}(t_{k+1} - \frac{|x-y|}{c})^2 \\ & + \int\int_{t_{k-1} \leq \frac{|x-y|}{c} \leq t_k} \frac{\vec{rot}\varphi_i(x)\vec{rot}\varphi_j(y)}{4\pi|x-y|}\frac{c^2}{2}(2\Delta t^2 - (t_{k+1} - \frac{|x-y|}{c})^2 - (t_{k-1} - \frac{|x-y|}{c})^2) \\ & + \int\int_{t_{k-2} \leq \frac{|x-y|}{c} \leq t_{k-1}} \frac{\vec{rot}\varphi_i(x)\vec{rot}\varphi_j(y)}{4\pi|x-y|}\frac{c^2}{2}(t_{k-2} - \frac{|x-y|}{c})^2 \\ \Big). & \end{aligned} \tag{A.20}$$

With $k < 0$ the integration domain is empty and the M^k matrices are nulls, this expresses the

fact that the test function $\varphi_i(x)\chi_m(t)$ does not interact with the functions $\varphi_j(x)\gamma_n(t)$ later in time ($n > m$). This is an essential property to solve the system since it allows a step by step solve in time.

A.7 Right Hand-side

The right hand-side writes

$$l_i^n = c \int_{t_{n-1} \leq t \leq t_n} \int_\Gamma \frac{\partial u_{inc}}{\partial n}(x,t).\varphi(x)dxdt. \tag{A.21}$$

In this formula, i is vertex id ($1 \leq i \leq N_S$) and $n \geq 1$ is a time step. If u_{inc} is a wave coming from the direction $\vec{r}$ (and not propagating in the direction $\vec{r}$), then it is written $u_{inc}(x,t) = f(t - t_0 + \vec{r}.\vec{x}/c)$ where f is the shape of the signal (Gaussian, sinusoid, ...) and t_0 is adjusting the time phase. We then get

$$\frac{\partial u_{inc}}{\partial n}(x,t) = \vec{n}.\vec{\nabla} u_{inc}(x,t) = \frac{\vec{n}.\vec{r}}{c} f'(t - t_0 + \frac{\vec{r}.\vec{0x}}{c}). \tag{A.22}$$

Consequently, l_i^n is given by

$$l_i^n = \int_\Gamma \vec{n}.\vec{r}\left[u_{inc}(x,t_n) - u_{inc}(x,t_{n-1})\right] \varphi(x)dx. \tag{A.23}$$

B

Mathematics

B.1 Polynomials

In the FMM-TD operator, in Section 5.1, we use the Legendre polynomial P_l and we use the Associate Legendre polynomial as a base for a second polynomial written Q_l^m.

B.1.1 Legendre Polynomial Formula

The definition of Legendre polynomial P_l is given in [109] by the following definition:

$$\begin{aligned} &(1-x^2)P_l''(x) - 2xP_l'(x) + l(l+1)P_l(x) = 0\,,\ l \in N \\ &P_l(1) = 1 \\ &P_l(-1) = (-1)^l\,. \end{aligned} \tag{B.1}$$

This formulation expresses the polynomial of order l based on its first and second derivative.

The computation of P_l can be obtain using the recurrence relation, $\forall l \in N$ and $\forall x \in [-1, 1]$:

$$\begin{aligned} P_0(x) &= 1 \\ P_1(x) &= x \\ P_l(x) &= \frac{2l-1}{l} x P_{l-1}(x) - \frac{l-1}{l} P_{l-2}(x),\ l \geq 2\,. \end{aligned} \tag{B.2}$$

A common alternative is to use the Rodrigues' formula which is also the basis of the associated Legendre polynomial, $\forall l \geq 0$ and $\forall x \in [-1, 1]$:

$$P_l(x) = \frac{1}{2^l l!} \frac{d^l}{dx^l} (x^2 - 1)^l \tag{B.3}$$

We remind that for all l we have

$$|P_l(x)| \leq 1\,.$$

B.1.2 Associated Legendre Polynomials

The associated Legendre polynomial is indexed by two variables l and m with $m \leq l$.

Using the Rodrigues' formula of the associated Legendre polynomial P_l^m, with $\forall (l, m) \in N^2$ and $0 \leq m \leq l$ we have:

$$P_l^m(x) = (-1)^m (1 - x^2)^{m/2} \frac{d^m}{dx^m} P_l(x)\,.$$

It can be extended to negative m (but still $|m| \leq l$) by

$$P_l^{-m}(x) = (-1)^m \frac{(l-m)!}{(l+m)!} P_l^m(x)\,.$$

In order to compute the polynomial values we can use the recurrence

$$\begin{cases} (l-m) P_l^m(x) = (2l-1) x P_{l-1}^m - (l+m-1) P_{l-2}^m \\ P_m^m(x) = (-1)^m (1 - x^2)^{m/2} (2m-1)!! \\ P_{m+1}^m(x) = (2m+1) x P_m^m(x)\,. \end{cases} \tag{B.4}$$

With $n!!$ the product of all even integers that are less than or equal to n.

This is written differently in [103]

$$\begin{cases} (l-m) P_l^m(x) - (2l-1) x P_{l-1}^m + (l+m-1) P_{l-2}^m = 0 \\ P_m^m(x) = (-1)^m (1 - x^2)^{m/2} \frac{(2m)!}{2^m m!} \\ P_{m+1}^m(x) = (2m+1) x P_m^m(x)\,. \end{cases} \tag{B.5}$$

The major difference is the expression of $(2m-1)!! = \frac{(2m)!}{2^m m!}$.

B.1.3 Q_l^m Polynomial

From [103; 104], we have the definition of the Q_l^m polynomial

$$\begin{cases} Y_{l,m}(\theta,\varphi) = C_{l,m}P_l^m(cos\theta)e^{im\varphi} \\ C_{l,m} = \sqrt{\frac{2l+1}{4\pi}\frac{(l-m)!}{(l+m)!}} \\ Q_l^m(cos\theta) = C_{l,m}P_l^m(cos\theta)\,. \end{cases} \tag{B.6}$$

where $Y_{l,m}$ are the spherical harmonics.

We can express Q_l^m by

$$\begin{cases} \sqrt{l^2-m^2}Q_l^m(x) - (2l-1)xQ_{l-1}^m + \sqrt{(l-1)^2-m^2}Q_{l-2}^m = 0\,, 0 \leq m \leq l \\ Q_m^m(x) = (-1)^m(1-x^2)^{m/2}\frac{\sqrt{(2m)!}}{2^m m!} \\ Q_{m+1}^m(x) = \sqrt{2m+1}xQ_m^m(x)\,. \end{cases} \tag{B.7}$$

This expression does not include the coefficient $\sqrt{\frac{2l+1}{4\pi}}$. For $m < 0$ we obtain by parity $Q_l^{-m}(x) = (-1)^m Q_l^m(x)$.

B.2 Unit Sphere Discretization

B.2.1 Basic Discretization

We remind that $Y_{l,m}$ can be written, with $C_{l,m}$ a constant, by

$$\begin{aligned} Y_{l,m}(\theta,\varphi) &= C_{l,m}P_l^m(cos\theta)e^{im\varphi} \\ C_{l,m} &= \sqrt{\frac{2l+1}{4\pi}\frac{(l-m)!}{(l+m)!}}\,. \end{aligned} \tag{B.8}$$

The discretization of the unit sphere has been taken from [103]. The objective is to find a grid of (φ_i,θ_j) to integrate exactly all the spherical harmonics until rank $2L$. Each pair has a weight given by $\omega_k = \omega_i^\varphi.\omega_j^\theta$.

For $(m,l) \neq (0,0)$ we have the condition

$$\begin{aligned} \sum_{k\in I}\omega_k Y_{l,m}(\vec{s_k}) &= \sum_{i,j}\omega_i^\theta.\omega_j^\varphi P_l^m(cos\theta_i)e^{im\varphi_j} \\ &= \left(\sum_i \omega_i^\theta P_l^m(cos\theta_i)\right)\left(\sum_j \omega_j^\varphi e^{im\varphi_j}\right) \\ &= 0\,. \end{aligned} \tag{B.9}$$

For φ we need a uniform distribution $\sum_j \omega_j^\varphi e^{im\varphi_j} = 0$ for $m \neq 0$ with $-2L \leq m \leq 2L$. We

choose a uniform distribution on $2L+1$ points on $[0, 2\pi]$

$$\begin{cases} \varphi_j = \frac{2\pi}{2L+1} j \,, 0 \leq j \leq 2L, \\ \omega_j^{\varphi} = \frac{2\pi}{2L+1} \,. \end{cases} \tag{B.10}$$

We obtain the geometric relation

$$\sum_{0 \leq j \leq 2L} \omega_j^{\varphi} e^{im\varphi_j} = \frac{2\pi}{2L+1} \sum_{0 \leq j \leq 2L} (e^{im\frac{2\pi}{2L+1}})^j = \frac{2\pi}{2L+1} \frac{1-(e^{im\frac{2\pi}{2L+1}})^{2L+1}}{1-(e^{im\frac{2\pi}{2L+1}})} \,. \tag{B.11}$$

The value is 0 if $m \neq 0$ and is 2π if $m = 0$.

In order to cover all the polynomials of degree $2L$ we take

$$\theta_i = \frac{\pi(i+1/2)}{2L+1}, 0 \leq i \leq 2L \,. \tag{B.12}$$

The weights are obtained by solving the linear system

$$\sum_{0 \leq i \leq 2L} P_l(cos\theta_i) \omega_i^{\theta} = \delta_{l,0} 2 \; \forall l = 0, ..., 2L \,. \tag{B.13}$$

The grid size is of dimension $(2L+1).(2L+1)$.

B.2.2 Advanced Approach

This method reduces the number of discretization points in the θ direction and is described in [110]. It gives $(L+1).(2L+1)$ Gauss points: $(L+1)$ θ and $(2L+1)$ φ which are similar to the previous method.

We first build the tridiagonal matrix T of dimension $L \times L$

$$\begin{cases} T_{i,i} = 0 \\ T_{i,i+1} = T_{i+1,i} = \frac{i}{\sqrt{4i^2-1}} \,, i = 1, ..., L-1 \,. \end{cases} \tag{B.14}$$

Then we compute the eigenvalues λ_i and their respective eigenvectors V_i. Finally, we obtain the different θ and their corresponding weights

$$\begin{cases} cos(\theta_i) = \lambda_i \\ \omega_i^{\theta} = 2(V_i)_0^2 \,. \end{cases} \tag{B.15}$$

B.3 Discret Fast Fourier Transform (DFFT)

We refer to [111; 112] to have more details on the underlying fast implementations.

B.3.1 Discrete Fourier Transform (DFT)

From a discrete signal f_n with data point at $n = 0, 1, 2, ..., N-1$ the discrete Fourier transform is given by

$$F_k = \sum_{n=0}^{N-1} f_n e^{-2i\pi nk/N} \,. \tag{B.16}$$

The inverse DFT is given by

$$f_n = \frac{1}{N} \sum_{k=0}^{N-1} F_k e^{2i\pi nk/N} \,. \tag{B.17}$$

From Euler's relation $e^{\pm ia} = cos(a) \pm isin(a)$, we have a possible implementation given by

$$\begin{aligned} F_k &= \sum_{n=0}^{N-1} f_n cos(k\omega_0 n) - if_n sin(k\omega_0 n) \\ f_n &= \frac{1}{N} \sum_{k=0}^{N-1} F_k cos(k\omega_0 n) + iF_k sin(k\omega_0 n) \,. \end{aligned} \tag{B.18}$$

The coefficient $\omega_0 = 2\pi/N$. However, this is a $0(N^2)$ algorithm and lower complexity methods have been proposed since.

B.3.2 Real DFT

When computing the DFT of a full real discrete signals we obtain a complex discrete results and we have

$$\begin{aligned} f &\to F \\ F(k) &= F^*(N/2 - k) \,. \end{aligned} \tag{B.19}$$

Here, $*$ stands for conjugate. This properties allow to optimize the DFT in time and space. When the DFT is from a real signal, we store only half of the resulting complex vector and if we need $F(k)$ with $k > N/2$ we compute the conjugate of $F(N/2 - k)$. This relation is used when computing back a complex signal, with inverse DFT, into a real signal.

B.3.3 Discrete Fast Fourier Transform (DFFT)

Many algorithms reduce the complexity of the DFT to $O(Nlog_2 N)$ as the Sande-Tukey or Cooley-Tukey algorithms. Extremely efficient libraries exist to compute DFT in real and complex numbers and one of the more advanced library is FFTW and has been presented in [108].

B.3.4 Convolution Product

The discrete convolution product is defined as

$$(f * g) \equiv \sum_{m=-\infty}^{+\infty} f[m]g[n-m] = \sum_{m=-\infty}^{+\infty} f[n-m]g[m]\,. \tag{B.20}$$

Both sources signal are discrete and have a finite number of values. If f has a values and g has b values, the result will have $a+b-1$ values. In term of discrete relation we have

$$(f * g)[i] = \sum_{j=max(0,i-b+1)}^{min(i,a-1)} f[j]g[i-j],\ 0 \leq i < a+b-1\,. \tag{B.21}$$

B.4 Differentiation

B.4.1 Derivatives of the Legendre Polynomial

The Legendre polynomial P_l is tied to its derivatives and we get from [113] the definition of the first and the second derivatives with the relation

$$(x^2-1)P_l'(x) = lxP_l(x) - lP_{l-1}(x)\,. \tag{B.22}$$

Which gives us

$$P_l'(x) = \frac{lxP_l(x) - lP_{l-1}(x)}{(x^2-1)}\,. \tag{B.23}$$

This is directly a recurrence relation which has the starting values $P_0'(x) = 0$ and $P_1'(x) = 1$.

The second derivative is expressed using Legendre and its first derivative

$$\begin{aligned} &(1-x^2)P_l''(x) - 2xP_l'(x) + l(l+1)P_l = 0 \\ &P_l''(x) = \frac{2xP_l'(x) - l(l+1)P_l}{(1-x^2)}\,. \end{aligned} \tag{B.24}$$

Again this is a recurrence formula where we have the starting values $P_0''(x) = 0$ and $P_1''(x) = 0$.

B.4.2 Taylor Series Differentiation

First derivatives.

$$
\begin{aligned}
f'(x_i) &= \frac{f(x_{i+1}) - f(x_i)}{h} + O(h) \\
f'(x_i) &= \frac{-f(x_{i+2}) + 4f(x_{i+1}) - 3f(x_i)}{2h} + O(h^2) \\
f'(x_i) &= \frac{f(x_i) - f(x_{i-1})}{h} + O(h) \\
f'(x_i) &= \frac{3f(x_i) - 4f(x_{i-1}) + f(x_{i-2})}{2h} + O(h^2) \\
f'(x_i) &= \frac{f(x_{i+1}) - f(x_{i-1})}{2h} + O(h^2) \\
f'(x_i) &= \frac{-f(x_{i+2}) + 8f(x_{i+1}) - 8f(x_{i-1}) + f(x_{i-2})}{12h} + O(h^4)\,.
\end{aligned}
\tag{B.25}
$$

Second derivatives.

$$
\begin{aligned}
f''(x_i) &= \frac{f(x_{i+3}) - 3f(x_{i+2}) + 3f(x_{i+1})}{h^3} + O(h) \\
f''(x_i) &= \frac{-3f(x_{i+4}) + 14f(x_{i+3}) - 24f(x_{i+2}) + 18f(x_{i+1}) - 5f(x_i)}{2h^3} + O(h^2) \\
f''(x_i) &= \frac{f(x_i) - 2f(x_{i-1}) + f(x_{i-2})}{h^2} + O(h) \\
f''(x_i) &= \frac{2f(x_i) - 5f(x_{i-1}) + 4f(x_{i-2}) - f(x_{i-3})}{h^2} + O(h^2) \\
f''(x_i) &= \frac{f(x_{i+1}) - 2f(x_i) + f(x_{i-1})}{2h} + O(h^2) \\
f''(x_i) &= \frac{-f(x_{i+2}) + 16f(x_{i+1}) - 30f(x_i) + 16f(x_{i-2}) - f(x_{i-2})}{12h} + O(h^4)\,.
\end{aligned}
\tag{B.26}
$$

B.5 Interpolation and Time Shifting

Having a signal $g(t)$ in the time domain, a time shifting leave the signal intact (same shape and scale) but shift it in the time direction. Shifting $g(t)$ by a time Δs gives

$$
f(t) = g(t - \Delta s)\,. \tag{B.27}
$$

With a discrete signal $g[t]$ the same formula is used if Δs is a multiple of Δx the time step between the discrete values. In this case, we move the values in a different positions. In case of a finite signal, a choice has to be made for the values outside the interval and it is usual to consider that they are zero. For a discrete finite signal of N values shifted by $s = \Delta s / \Delta t$

$$
\begin{aligned}
f[t] &= g[t - s]\ \forall 0 \leq s < N \\
f[t] &= 0\ \textit{else}\,.
\end{aligned}
\tag{B.28}
$$

But in most cases the shift is not a multiple of Δt and thus the values written in $f(t)$ has to be interpolated from the values in $g(t)$.

B.5.1 Linear Interpolation

The more basic interpolation is to compute the shifted value using two discretized values and the appropriate weights. From a discrete signal g of time step Δt and length N and a shift Δs the result is given by

$$f[t] = g[t-s](1-c) + g[t-s+1]c,\ 0 \leq t-s+1 < N\,. \tag{B.29}$$

Here, s is the largest integral value not greater than $\Delta s/\Delta t$ and $c = s - \Delta s/\Delta t$ is a coefficient $\in [0,1]$ and is the weight to balance between the two source values.

B.5.2 Cubic Spline Interpolation

A spline is composed of piecewise continuous polynomial functions, it is of degree n if it is composed of polynomials of degree n. Each polynomial is used to connect two contiguous points. A linear spline connect the discrete points using straight lines, a quadratic spline uses quadratic polynomials etc. To ensure a smooth aspect between each polynomials that connect the same point, they must have the same derivative up to a given degree. In a quadratic spline the polynomials must have the same first derivative and in a cubic spline the polynomials must also have the same second derivative. From N points (x_k, y_k) with $0 \leq k < N$, and S_k the kth polynomials, we have $S_k(x_k) = y_k$. For the continuity we have $S_k(x_{k+1}) = S_{k+1}(x_{k+1})$ and the equality implies also for the derivatives: $S'_k(x_{k+1}) = S'_{k+1}(x_{k+1})$ and in case of cubic spline $S''_k(x_{k+1}) = S''_{k+1}(x_{k+1})$. But finding the coefficients of the polynomials requires to solve a sparse linear system that is why we use Hermitian spline. Among the general cubic spline, a Hermitian spline refers to the single cubic polynomial.

Having two point (x_0, p_0) and (x_1, p_1) and knowing the derivatives on these points m_0 and m_1 respectively, a Hermitian spline is a cubic polynomial P with $P(x_0) = p_0$, $P(x_1) = p_1$, $P'(x_0) = m_0$ and $P'(x_1) = m_1$. This polynomial is given by

$$\begin{aligned}
h_{00}(t) &= 2t^3 - 3t^2 + 1 \\
h_{10}(t) &= t^3 - 2t^2 + t \\
h_{01}(t) &= (-2)t^3 + 3t^2 \\
h_{11}(t) &= t^3 - t^2 \\
P(t) &= h_{00}(t)p_0 + h_{10}(t)m_0 + h_{01}(t)p_1 + h_{11}(t)m_1\,.
\end{aligned} \tag{B.30}$$

The derivatives of discrete points can be obtained from Taylor expansions.

B.5.3 Interpolation with FFT

There is a relation between a time shifting of a real function and its Fourier transform

$$f(t - t_0) \Leftrightarrow F(k)e^{2\pi ikt_0} \tag{B.31}$$

C

Hardware Overview

C.1 Modern CPU

The development of high-performance application must take into account the hardware specificities of the CPU. The key-points of modern CPU are the hierarchical memory structure and the management of instructions; how the instructions are fetched and pipelined. We divide the CPU in three parts which are shown in Figure C.1: the memory parts (composed by the buses and the memory levels), the instruction part (the control unit) and the computation part (the execution unit).

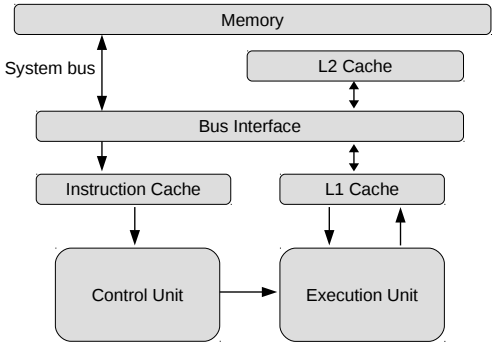


Figure C.1: CPU Schematic View

Memory hierarchy. The fast memory is expensive, and moreover, it has to be close to the processing units such that it is not viable to build a CPU with only this kind of memory. On the other hand, the slow memory is cheap and can be incorporated in large quantity in a chip. Therefore, the current strategy to hide the limited fast memory is to use a hierarchical memory with different layers; from small and fast to large and slow. The levels between the processing units and the *real* memory are called caches and their names are postfixed by the distance from the processing units. It is usual to have 3 levels of cache named $L1$, $L2$ and $L3$. This hardware structure makes the development of application difficult because the developers have very low control over the memory execution. While there exists some instructions to prefetch some data to a given layer or to flush the caches, using these optimizations is difficult because the developers have a very limited view on what happen in the memory and cannot really control the data movement. Moreover, several hardware modules are responsible for the memory control and management which leads to unpredictable behaviors. Therefore, developing an efficient application is not done by an explicit memory control but by a development based on the hardware understanding and modeling.

During a program execution, the memory transfers are driven by two properties which are the latency and the bandwidth. The latency is the time it takes for some Bytes requested from the memory to arrive at the destination. The bandwidth is the number of Bytes that arrived together by a data movement. If the latency is l and the bandwidth is b then it takes $\approx l + n/b$ to receive n Bytes. The latency of modern CPUs is about $1ns$ for $L1$, $3 - 10ns$ for $L2$, $10 - 20ns$ for $L3$ and $50 - 100ns$ for the main memory. The bandwidth is around $50GB$ per second for a CPU with four channels.

Control Unit. The control unit is responsible for the program execution: what has to be executed and when. It retrieves instructions from the program memory, decodes the instructions, retrieves the data from memory and stores the results. The performance of an execution is driven by the capacities of this module. It is now widespread for a control unit to support instruction prefetch and decoding (pipelining), branch prediction, out of order execution and retirement. To prefetch means to fetch (retrieve) a data or an instruction before they are needed by looking in advance at what will be used. The decoder prepares data and instructions in a cache because it considers that they will be used by the execution unit in a near future. However, a branch might lead to miss prediction and in this case all the prepared material may be cleared. To reduce the number of branch miss-predictions, the branch prediction unit uses advanced techniques as Deep branch prediction (attempting to decode multiple paths), Dynamic data flow analysis and Speculative execution. The out-of-order engine is able to analyze a large number of instructions and to find the independent ones that can be executed. Then these different paths are saved using allocator, potentially modified by register renaming and managed by micro operation scheduler. The allocator unit ensures that a buffer space is allocated properly for each instruction. The register renaming module uses logical registers to process the instructions: instead of the general-purpose registers, it maps a real register with a logical register which allows simultaneous access to the same register by multiple instructions. The mapping is kept in the RAT register allocation table. The micro operation sched-

uler determines if a micro operation is ready, which means that the input elements are ready, and it maintains the program dependencies using several queues and ports.

The pipelining of the operations accelerates the execution. Without pipelining, if it takes L steps to prepare and compute an instruction then it takes $n.L$ to compute n instructions. If these steps are independent because they work on different data they can be pipelined. The objective is to keep the different steps busy by starting to process an instruction in the first stage without waiting for the previous one to be completed. Such that with the pipelining, it takes $n + L - 1$ cycles to compute n tasks. The pipelining is helped by the speculative mechanism which finds several instructions that can be executed at the same time or what will be the next instructions. Therefore, even so the current instruction is a test which might change the execution path, the speculative module gives a possible next instruction and somehow it allows to compute entirely or partially a result. However, the prediction may fail and in this case the speculative operation would have been done for nothing. These several mechanisms belong to the instruction-level parallelism (ILP) techniques. While they are managed by hardware modules, the Assembly instructions and thus the software-development side drastically influences their success.

Execution units. These modules execute the instructions and there might be several execution units per processor working simultaneously. It is usual to have separated arithmetic and logic unit (ALU) each dedicated to a specific type of operations: simple integer operations, complex integer operations and floating-point operations. The boolean operations and simple integer operations are done with low latency. The ALU for complex integer operations is responsible for the multiplications, divisions and rotations of integer data type. The floating-point module supports operations on the floating-point types (single and double) and it is common for this module to support the SIMD. The general-purpose registers store temporary data needed by the processor coming from computation results or from the main memory and their usage is a good help to manage the instructions and the memory accesses.

Cache policies. There are usually 2 or 3 cache levels, and we talk of a cache block/line to define the data copied from a upper cache into a lower level. When a part of the memory is requested by an operation, it is loaded in the cache (if it is not already in it). More precisely, the cache line which includes the requested address is stored in a deterministic part of the cache from a mapping algorithm. The mapping is direct if there is only one place where the data go based on their addresses, or set associative if there are k places where the data can go (it is said to be a k-way and if k=1 it is a direct mapping). It is usual to use $4 - way$ or $8 - way$ set associative caches. The associative algorithm to find the position for a cache line can be pseudo random, but it is clear that contiguous data in memory should not use the same set. By definition the caches are smaller than their respective upper-level memory, such that when a data is moved inside, we have to remove another cache line. Inside a set, the replacement policy can be random too, but most CPU use a variation of least recently used (LRU) algorithm; a cache line replaces the line in the set which has not been used since the longest time. The objective is to have temporal locality and to reduce the

cache misses (asking for a memory that is not in the cache) because the request will need to check upper-levels and finally be more than 100 times slower.

If a data in the caches is modified, the system uses a policy to decide when the change is propagated to the main memory. A write-through system directly writes back the modified block into the memory; with this strategy, the main memory is always up to date with the latest modified version in it. In the write-back strategy, a block is saved if it is pulled from the cache and is a modified version of the main memory, or if it is loaded by another cache. To ensure cache coherence (when data are written back, what to do if a data exists in multiple cache): the main approach is snooping cache coherence. But it is also possible to use direct based cache coherence: in this protocol a directory is maintained up to date and store the status of each cache line (which caches are currently holding a copy of the modified line). Several threads should not work on data that are on the same cache line or it will imply false sharing; when a thread modifies a part of the line which is not used by the other thread, the coherency protocol will still need to propagate the change.

Some special instructions load/prefetch non-temporal data into the lower cache and bypass the intermediate caches. They should be used when data is access during a short period (or at least that they should not be kept in the cache because we know they will be evicted/removed before their next utilization). The hardware makes special optimization by knowing that some data should not be kept in the cache.

Non-uniform memory access (NUMA). NUMA is a memory design where the shared memory is distributed on the different processors of a machine with the consequence of a variable data-access time. A delay (latency) comes from the distance between the processors and the memory hosts. Especially, it is faster for a processor to ask for a data from its own memory rather than for a data located in a *remote* memory. Cache coherency with a NUMA memory is even more complicated because it adds an additional memory level. Dealing with NUMA effect is not an easy task, however to reduce the effect, the allocation of the data should be distributed on the different memory node and for example, when the threads are bound to a core, they should allocate their data on their respective node. But when a large block is shared by all the threads, or if the execution choices are made dynamically it is difficult to predict the good memory mapping.

Locality. The different modules and their respective behaviors lead to two different localities. The temporal locality could be defined by: *A data which is used now will certainly be used again*. This property is optimized by the LRU and the multi-layers memory. The spatial locality could be defined by: *The memory neighbor of a data that is used now will certainly be used too*. This property is ensured by the size of cache lines and the possible prefetch (load in advance) of next closest values.

C.2 Single Instruction, Multiple Data (SIMD/SSE/AVX)

The processor frequencies have stopped increasing for several years but the manufacturers continue to push the peak performance by using multiprocessors, ILP and SIMD. The term SIMD

refers to the concept of having multiple processing elements which perform the same operation on multiple data simultaneously. The SIMD is a model introduced by the Flynn's taxonomy classification where we also find SISD, MISD and MIMD. Historically, the first SIMD processors were classified as vector-processors such that it is now common to use the word *vectorization* to refer to the SIMD instruction sets (SSE/AVX). The usage of SIMD was motivated by the matrix operation algorithms which are used by numerous applications but it is now applied to a width range of domains (including non-numerical applications). The current SIMD instructions operate on a contiguous memory block which allows some energy-efficiency properties and facilitates the memory transfer. In the rest of the discussion, the term SIMD refers to the current implementations (SSE/AVX). The SIMD instructions operate on objects of a defined size S; the SSE standard deals with $S_{SSE} = 128$ bit words and the AVX with $S_{AVX} = 256$ bit words. This SIMD word is divided into M scalar of a native data type, for example in SSE it contains $M = 4$ values of 32 bits like *int*, *float* or $M = 2$ values of 64 bits like *double*. The SIMD instruction unit can perform usual arithmetic computation (addition/subtraction/multiplication/division) but also advanced arithmetic operations (square root), special functions (minimum, maximum) or bit/word manipulations (bit shifts, bits arithmetic). A classic SIMD operation *op* is applied term-to-term on two SIMD typed word a and b as $c(i) = a(i)\, op\, b(i)$ for $\forall i \in [1; M]$. The SIMD words are stored in the vector registers which are now equivalent to the general floating-point registers. The SIMD instruction sets provide operands to manage the memory accesses to load one or several values or to save an SIMD word into the main memory. The memory alignment is crucial to achieve high performance and it is usual that the alignment is a multiple of S. If the memory is not aligned many instructions are forbidden and should be replaced by more soft but slower instructions. There is a class of very common operations (as the *horizontal* sum) that are not supported natively by the SIMD specifications and that sometime makes the development complicated to perform basic tasks. For example, there is no instruction to merge/reduce all the values contained in an SIMD word into a scalar value. Developing a code which takes advantage of the vectorization of the processors is mainly done by four techniques:

- programming in a high level language with scalar values and delegate to the compiler the possible optimizations as presented in Section C.2.1.
- programming in a high level language with scalar values and give hints to the compiler regarding the possible optimizations using *pragma* like OpenMP 4 SIMD specification
- programming in a high level language using intrinsic instructions and delegate the conversion into Assembly to the compiler as presented in Section C.2.2.
- programming directly in Assembly and specify the SIMD operations and registers as presented in Section C.3.

C.2.1 UBCOO SpMV in *C++*

Most of the compiled languages have the potential to use the SIMD units of a CPU but this capacity relies on the *understanding* of the code by the compiler and the type of algorithm. In fact,

many algorithms are easily compatible with the SIMD structure which deals with contiguous data . In the Code C.1, we present a *C++* kernel to compute the UBCOO SpMV introduced in Section 3.1.5. Without any specific optimizations, the compiler will create a binary which works on most machines and will not use the SIMD capability. With Gcc, we can enable the use of certain instruction sets (using direct flags like $-msse$ or $-mavx$) or by specifying the target architecture $-march = X$. Moreover, by activating a high level of optimization, the compiler is able to unroll loops and to adapt them in order to use SIMD instructions. In the code snippets, the size of the block is templatized and is known at compile time. In addition, the leading dimension of the *blockValue* is equal to *BlockDim* and is also known at compile time and the data accesses are linear and regular such that the compiler should be able to use the SIMD appropriately. But as it is shown in Section 3.1.6, the given results are disappointing and the compiler is not able to create an efficient binary.

```
template <const int BlockDim, class ValueType>
void BlockSpMV(const ValueType blocks[], const int blocksIdxs[], const int nbBlocks,
               const ValueType x[], ValueType y[]){
    for(int idxBlock = 0 ; idxBlock < nbBlocks ; ++idxBlock){
        ValueType* __restrict__ ptrY = &y[blocksIdxs[idxBlock*2]];
        const ValueType* __restrict__ ptrX = &x[blocksIdxs[idxBlock*2+1]];
        const ValueType* __restrict__ blockValue = &blocks[idxBlock*BlockDim*BlockDim];

        for(int idxCol = 0 ; idxCol < BlockDim ; ++idxCol){
            for(int idxRow = 0 ; idxRow < BlockDim ; ++idxRow){
                ptrY[idxRow] += ptrX[idxCol] * blockValue[idxCol*BlockDim + idxRow];
            }
        }
    }
}

```

Code C.1: SpMV of block size 8×8 in C++

C.2.2 UBCOO SpMV in *C++* using Intrinsics

An intrinsic is an operand handled directly by the compiler which usually substitute the call by a given code/instruction. The SIMD Assembly operands are usable using intrinsics and special data types. Many documents and websites summarized the intrinsics but a great tool is hosted by Intel Intrinsics Guide [114]. The SIMD intrinsic functions are named according to a logic [*size*]_[*operand*]_[*datatype*], where size is *_mm* (SSE 128 bits), *_mm*256 (AVX 256 bits) or *_mm*512 (AVX2 512 bits). In the Code C.2, we show an example of our UBCOO SpMV using *AVX* intrinsics in *C++*. This code is for blocks of size 8×8 and we could have templatized the number of columns in the block. We also could have templatized the number rows in the blocks (to a multiple of 8) but it may not give the same performance. Here we consider that the memory address of the matrix (*blocks*) is 64 Bytes aligned which allows to use faster load operands. From the performance result in Section 3.1.6 this kernel is achieving a good Flop-rate. Developing a kernel in intrinsic is usually sufficient to achieve the maximum performance regarding the Flop/word ratio. However, while we can use the *__register__* keyword to advise the compiler to store some variable in the registers or the

__restrict__ keyword to avoid aliasing, the compiler is still in charge of most of the optimizations.

```
void BlockSpMV_8_avx(const double blocks[], const int blocksIdx[], const int nbBlocks,
                     const double x[], double y[]){
    for(int idxBlock = 0 ; idxBlock < nbBlocks ; ++idxBlock){
        double* __restrict__ ptrY = &y[blocksIdx[idxBlock*2]];
        const double* __restrict__ ptrX = &x[blocksIdx[idxBlock*2+1]];
        const double* blockValue = &blocks[idxBlock*8*8];

        __m256d res03 = _mm256_setzero_pd();
        __m256d res47 = _mm256_setzero_pd();

        for(int idxCol = 0 ; idxCol < 8 ; ++idxCol){
            const __m256d colVal = _mm256_set1_pd(ptrX[idxCol]);

            res03 = _mm256_fmadd_pd(colVal, _mm256_load_pd(&blockValue[idxCol*8]), res03);
            res47 = _mm256_fmadd_pd(colVal, _mm256_load_pd(&blockValue[idxCol*8+4]), res47);
            // Or if fmadd is not available:
            // res03 = _mm256_add_pd(_mm256_mul_pd(colVal, ...
                        _mm256_load_pd(&blockValue[idxCol*8])), res03);
            // res47 = _mm256_add_pd(_mm256_mul_pd(colVal, ...
                        _mm256_load_pd(&blockValue[idxCol*8+4])), res47);
        }

        _mm256_storeu_pd(ptrY, _mm256_add_pd(_mm256_loadu_pd(ptrY), res03));
        _mm256_storeu_pd(ptrY+4, _mm256_add_pd(_mm256_loadu_pd(ptrY+4), res47));
    }
}

```

Code C.2: SpMV of block size 8×8 in AVX intrinsics

C.3 $x86-64$ Assembly

When the code is written in *C/C++* many optimizations are done by the compiler when the code is transformed into Assembly. It is difficult to influence the resulting binary because it is a mix of what the compiler understands of the code, how it applies optimizations, what the developer understands of the compiler optimizations and the hardware and how the *C/C++* code is written. It is difficult to help the compiler and it may result in anti-optimizations because of miss-understanding between what the developer tries to express and what the compiler understands. This motivates the use of Assembly which let interact directly with the hardware.

The $x86-64$ specification has been proposed by AMD (*AMD64*) as an extension of the $x86$ and is also supported by many manufacturers including Intel (*EM64T*). It aims at extending the $x86$ to 64 bits architecture with the support of long int and 8 Bytes pointers. But it also doubles the number of general-purpose registers (16 instead of 8) and XMM/SIMD registers (16 instead of 8) and it adds some advanced features like the passage of function parameters using registers. The general-purpose registers manipulate types of 1, 4 or 8 Bytes according to the operand names. For example, the *mov* operand should be postfixed like $mov\{l, q, w\}$ to specify the size of the data and followed by an $x86$ register (%rax, %rcx, %rdx, %rbx, %rsi, %rdi, %rsp, and %rbp) or a new register (from %r8 to %r15). The registers have also a main name which can be postfixed to

match the target data type. Intel and AMD provide great documentations [115; 116] which define instructions set and also give some optimization advices.

Developing in Assembly appears to be a difficult task at first sight. However, in Scientific applications the $80-20$ rule is usually true and 80% of the time is spent in 20% of the code. Therefore, it is interesting to invest in order to optimize the computation intensive parts which are usually included in small kernels. For example, most BLAS or sparse BLAS functions are written in a few lines (<100). Moreover, these computational kernels are adapted to be directly expressed in Assembly because

- they do not call external functions, all the code is included in the body of the function
- they do not access structures or classes or any complex memory element
- they declare only pointers and indexes
- they simply iterate linearly or by indirection with one or several loops
- they are a translation of the *C* with SIMD intrinsics without the potential extra instruction added by the compiler
- they do not allocate any memory or I/O
- they compute floating-point operations; they load values from the memory, compute and store back the results
- they do not need to access/modify the stack especially in $x86-64$ because the function parameters are passed by registers.

It is accessible to convert a code written in standard *C* into *C* + *intrinsics* but it is the same principle to convert from *C* + *intrinsics* into Assembly. The readability of the Assembly makes it less intuitive but the structure of the kernels and the important operands are easy to master. We spread for a register-oriented programming with the ideas of maximizing the usage of the registers, avoiding the usage of the stack and minimizing the number of loads. A compiler is able to create such Assembly especially from a *C*+*intrinsics* code, but in general they delegate many optimizations to the hardware. Moreover, this kind of development is not a general programming model and it cannot be used for general-purpose applications. We provide an example of an Assembly kernel in the Code C.3 to compute our UBCOO SpMV presented in Section 3.1.6.

```
1  extern "C" void BlockSpMV_8_avx_asm(const double* blocks, const int* blocksIdx, const size_t …
       nbBlocks,
2                                      const double* x, double* y);
3  /* (blocks rdi, blocksIdx rsi, nbBlocks rdx, x rcx, y r8) */
4  /* use r9 for block row idx, r10 for block col idx, r13/r14/r15 unused */
5  __asm__(
6      ".global BlockSpMV_8_avx_asm \n"
7      "BlockSpMV_8_avx_asm: \n"
8      "SpSliceAvxAsm_gmvv_mod4_f_startloop:\n"
9      "PREFETCHNTA 1024(%rdi);\n"
10     "movl 0(%rsi),%r9d; \n"                    // blocksIdx[0], row idx
11     "movl 4(%rsi),%r10d; \n"                   // blocksIdx[1], col idx
12     "lea (%rcx,%r10,8),%r10; \n"               // r10 = rcx +r10 = x + blocksIdx[1] ; …
       blocksIdx[1] * sizeof(double), col idx
13
14     "vmovupd 0(%r8, %r9, 8), %ymm0; \n"        // y[rowIdx]
```

```
    "vmovupd 32(%r8, %r9, 8), %ymm1; \n"        // y[rowIdx+4]

    // Fill all ymm values with vbroadcastsd or vshufpd
    "vbroadcastsd 0(%r10), %ymm2; \n"           // ptrx[0]
    "vmovapd (%rdi), %ymm3; \n"                 // values[0][0] / [0][0]
    "vmovapd 32(%rdi), %ymm4; \n"               // values[1][0] / [4][0]
    "vfmadd231pd %ymm3, %ymm2, %ymm0; \n"       // res0 += ptrx[0] * values[0][0]
    "vfmadd231pd %ymm4, %ymm2, %ymm1; \n"       // res1 += ptrx[0] * values[1][0]

    "vbroadcastsd 8(%r10), %ymm5; \n"           // ptrx[1]
    "vmovapd 64(%rdi), %ymm6; \n"               // values[0][1] / [0][4]
    "vmovapd 96(%rdi), %ymm7; \n"               // values[1][1] / [4][4]
    "vfmadd231pd %ymm6, %ymm5, %ymm0; \n"       // res0 += ptrx[0] * values[0][0]
    "vfmadd231pd %ymm7, %ymm5, %ymm1; \n"       // res1 += ptrx[0] * values[1][0]

    "vbroadcastsd 16(%r10), %ymm8; \n"          // ptrx[2]
    "vmovapd 128(%rdi), %ymm9; \n"              // values[0][2] / [0][8]
    "vmovapd 160(%rdi), %ymm10; \n"             // values[1][2] / [4][8]
    "vfmadd231pd %ymm9, %ymm8, %ymm0; \n"       // res0 += ptrx[0] * values[0][0]
    "vfmadd231pd %ymm10, %ymm8, %ymm1; \n"      // res1 += ptrx[0] * values[1][0]

    "vbroadcastsd 24(%r10), %ymm11; \n"         // ptrx[3]
    "vmovapd 192(%rdi), %ymm12; \n"             // values[0][3] / [0][12]
    "vmovapd 224(%rdi), %ymm13; \n"             // values[1][3] / [4][12]
    "vfmadd231pd %ymm12, %ymm11, %ymm0; \n"     // res0 += ptrx[0] * values[0][0]
    "vfmadd231pd %ymm13, %ymm11, %ymm1; \n"     // res1 += ptrx[0] * values[1][0]

    "vbroadcastsd 32(%r10), %ymm2; \n"          // ptrx[4]
    "vmovapd 256(%rdi), %ymm3; \n"              // values[0][4] / [0][16]
    "vmovapd 288(%rdi), %ymm4; \n"              // values[1][4] / [4][16]
    "vfmadd231pd %ymm3, %ymm2, %ymm0; \n"       // res0 += ptrx[0] * values[0][0]
    "vfmadd231pd %ymm4, %ymm2, %ymm1; \n"       // res1 += ptrx[0] * values[1][0]

    "vbroadcastsd 40(%r10), %ymm5; \n"          // ptrx[5]
    "vmovapd 320(%rdi), %ymm6; \n"              // values[0][5] / [0][20]
    "vmovapd 352(%rdi), %ymm7; \n"              // values[1][5] / [4][20]
    "vfmadd231pd %ymm6, %ymm5, %ymm0; \n"       // res0 += ptrx[0] * values[0][0]
    "vfmadd231pd %ymm7, %ymm5, %ymm1; \n"       // res1 += ptrx[0] * values[1][0]

    "vbroadcastsd 48(%r10), %ymm8; \n"          // ptrx[6]
    "vmovapd 384(%rdi), %ymm9; \n"              // values[0][6] / [0][24]
    "vmovapd 416(%rdi), %ymm10; \n"             // values[1][6] / [4][24]
    "vfmadd231pd %ymm9, %ymm8, %ymm0; \n"       // res0 += ptrx[0] * values[0][0]
    "vfmadd231pd %ymm10, %ymm8, %ymm1; \n"      // res1 += ptrx[0] * values[1][0]

    "vbroadcastsd 56(%r10), %ymm11; \n"         // ptrx[7]
    "vmovapd 448(%rdi), %ymm12; \n"             // values[0][7] / [0][28]
    "vmovapd 480(%rdi), %ymm13; \n"             // values[1][7] / [4][28]
    "vfmadd231pd %ymm12, %ymm11, %ymm0; \n"     // res0 += ptrx[0] * values[0][0]
    "vfmadd231pd %ymm13, %ymm11, %ymm1; \n"     // res1 += ptrx[0] * values[1][0]

    "vmovupd %ymm0, 0(%r8, %r9, 8); \n"         // y[rowIdx]
    "vmovupd %ymm1, 32(%r8, %r9, 8); \n"        // y[rowIdx]

    "addq $512, %rdi;\n"                        // blocks += 8*8 (*8)
    "addq $8, %rsi;\n"                          // blocksIdx += 2 (*4)

    "subq $1, %rdx;\n"                          // nbBlocks -= 1
    "jnz SpSliceAvxAsm_gmvv_mod4_f_startloop;\n" // if nbCols != 0 go to beginning

```

```
75      "NOP;\n" // To avoid the presence of ret just after the jnz
76
77      "BlockSpMV_8_avx_asm_out:\n"
78      "ret;\n"
79  );
80
```

Code C.3: SpMV of block size 8×8 in Assembly x86-64

C.4 NVidia GPUs and CUDA Programming

The graphical processing units (GPUs) have been improved drastically for 20 years principally pushed by the video-game industry. The usage of the GPUs to perform scientific computing was more a hack before NVidia introduced its CUDA programming model [9]. It has been the real beginning of the general-purpose computation on graphical processing units (GPGPU). The GPUs are classified in the family of the co-processors which includes processing units that are responsible for a specific task but cannot work without the CPU; the CPU is said to be the host and the GPU the device. The CPU is in charge of the allocations and copies to the device but also to the calls to the GPU kernel functions (functions that are executed on the device). However, the GPUs capabilities continue to be improved and the latest CUDA release offers new possibilities like dynamic/nested parallelism. The GPU hardware specificities and the runtime parameters are critical to drive the development of efficient kernels.

A GPU is composed by several processors and each of them computes several thread-blocks. A processor could be described as an SIMD/vector unit because it can perform a single instruction on multiple data but in reality, the term SIMT (Single-Instruction, Multiple-Thread) is used to avoid the confusion with CPU SIMD. The width of the SIMT stream is called the warp and is equal to 32 on most GPUs. A processor creates, manages, schedules, and executes threads in a team of 32 parallel threads called warps. The memory system is composed by a global/main memory, a shared memory, a local memory and the registers.

The CUDA programming model has nicely hidden the complexity of the GPUs by providing a multi-thread oriented programming even though the execution is closer to an SIMD. This abstraction is ambiguous at first sight and leads to poor performance if a kernel is developed like for a multi-thread CPU. In CUDA, we create a multi-dimensional grid of thread-blocks where each thread-blocks is a multi-dimensional thread-team. The model is mapped above the GPU hardware and the thread-blocks are distributed among the processors and the threads are executed by team of warp. The threads that composed a warp start together at the same program address, but they have their own instruction address counter and register state and are therefore free to branch and execute independently. Therefore, even so the term thread is used, this has nothing to do with a CPU/OS thread but it allows to represent the different units of the SIMT processors. If the threads inside a warp do not execute the same instruction, each different instruction is done separately and one after the other.

On the CPU, the memory control is very limited but on the GPU a part of the memory manage-

ment must be done by the programmer. Each memory level has its own specificities in terms of visibility and access. The global memory is visible by all the threads, the shared memory is visible by the workers of a thread-block and the local memory and the registers are visible by one thread. The access to the global memory is performed transparently with a pointer data type but it relies on a complex mechanism. The accesses should be coalesced/aligned because in a simplified model we consider that each 128 Bytes word is read separately. The shared memory is faster but very limited and belongs to a thread-block, the access to the shared memory should be strided because the addresses are mapped to different banks and several accesses to the same bank to different memory addresses are done consecutively. The data of a thread are stored into registers and if it is not enough into the local memory. The registers are extremely fast and should be privileged such that a kernel does not need to use the local memory.

The size of the different memories is also crucial in the development of the kernels. The main GPUs limitations are their number of processors, the maximum size of the grid, the maximum size of the thread-blocks, the size of the shared memory, the maximum number of registers per thread-blocks/threads. Moreover, to achieve high performance the processors must be occupied which involves having a large number of threads because the instructions are pipelined at the instruction-level parallelism within a single thread, but also at a thread-level parallelism. In Code C.4, we provide the source code of our kernel for the slice computation presented in Section 3.4.3 where we try to take into account all the specificities of the GPU to get an efficient implementation.

```
template <class ValueClass, const int nbMergedSteps,
                            const int matrixBlockDimWidth, const int maxNbNnzMat>

__device__ void cuda_compute_a_rowgroup(
           const int leadingRes, const int nbRows,
           const int* startingPointPerRow, const int nbThreadsInGroup,
           const ValueClass* values, const ValueClass* pastValues,
           ValueClass* results, const int nbNnzMat) {

    // Copy of the past values related to the current slice
    __shared__ ValueClass sharedPast[maxNbNnzMat + nbMergedSteps];

    for(int idx = threadIdx.x ; idx < (nbNnzMat+nbMergedSteps) ; idx += blockDim.x){
        sharedPast[idx] = pastValues[idx];
    }
    __syncthreads();

    // Computation of the Contiguous Block
    values += threadIdx.x;
    for(int idxRow = threadIdx.x ; idxRow < nbRows ; idxRow += blockDim.x){
        const ValueClass* ptrPast  = &sharedPast[startingPointPerRow[idxRow]];
        ValueClass sum[nbMergedSteps];

        {
            const int idxCol = 0;
            const ValueClass colValue = values[idxCol];
            #pragma unroll
            for(int idxRes = 0 ; idxRes < nbMergedSteps ; ++idxRes){
                sum[idxRes] = ptrPast[idxCol+idxRes] * colValue;
            }
        }
```

```
32          #pragma unroll
33          for(int idxCol = 1; idxCol < matrixBlockDimWidth; ++idxCol){
34              const ValueClass colValue = values[idxCol*nbRows];
35              #pragma unroll
36              for(int idxRes = 0 ; idxRes < nbMergedSteps ; ++idxRes){
37                  sum[idxRes] += ptrPast[idxCol+idxRes] * colValue;
38              }
39          }
40
41          ValueClass* ptrRes          = &results[idxRow];
42          ptrRes[0] += sum[0];
43          #pragma unroll
44          for(int idxRes = 1 ; idxRes < nbMergedSteps ; ++idxRes){
45              ptrRes[idxRes*leadingRes] += sum[idxRes];
46          }
47
48          values += blockDim.x;
49      }
50
51      // Barrier to ensure that no thread goes to the next slice
52      __syncthreads();
53  }
54
```

Code C.4: Contiguous-Blocking CUDA source code

D

FMM

D.1 Hilbert Indexing

The conversion of the box/grid-coordinate into Hilbert index is not trivial, and a nice algorithm has been proposed in [117]. We give the method in Algorithm 17, where the rotations and the inversions are done in a single switch statement.

D.2 Morton Indexing

Figure D.1 shows how the bits of the grid-coordinate are interleaved to give a Morton index. The example is for $3D$ coordinate but the same principle applies to any dimension.

$$x = x_{B-1}x_{B-2}...x_1x_0 \; ; \; y = y_{B-1}y_{B-2}...y_1y_0 \; ; \; z = z_{B-1}z_{B-2}...z_1z_0$$
$$mindex = x_{B-1}y_{B-1}z_{B-1}.x_{B-2}y_{B-2}z_{B-2}....x_1y_1z_1.x_0y_0z_0$$

Figure D.1: Morton Index example in $3D$

Therefore, the conversion is done by performing one interleaving per level as shown in Algorithm 18. The complexity of this algorithm is $O(l)$ with l the level of the target cell.

The Algorithm 19 reverses the conversion and return a grid-coordinate from a Morton index. The complexity of this algorithm is $O(l)$ with l the level of the target cell.

Algorithm 17: Grid-Coordinate to Hilbert index

```
Data: C_m the position of the cell, l the level of the cell
function GetHilbertIndex(C_m, l) : hindex
    for idxl = l-1 to 0 do
        // Get the bit at position idxl from integer x
        xi = bits_get_bit(x, idxl);
        yi = bits_get_bit(y, idxl);
        zi = bits_get_bit(z, idxl);
        index = bits_left_shift(xi,2) OR bits_left_shift(yi,1) OR zi;
        switch index do
            case 000_B
                swap(y, z);
                break;
            end
            case 001_B or 101_B
                swap(x, y);
                break;
            end
            case 100_B
                bits_inverse(x);
                bits_inverse(z);
                break;
            end
            case 110_B
                bits_inverse(x);
                bits_inverse(z);
                break;
            end
            case 111_B or 011_B
                swap(bits_inverse(x), bits_inverse(y));
                break;
            end
            otherwise
                // index is 010_B
                swap(bits_inverse(y), bits_inverse(z));
            end
        endsw
        hindex = bits_left_shift(hindex,3) + transform[index];
    end
    return hindex;
```

D.2.1 Interaction List Computation

In the FMM far-field, we have to find the neighbors of the leaves ($P2P$) and the cell interaction lists ($M2L$). We recall that the interaction list for a given cell c at level l is composed by the children of the neighbors of c's parent who are not direct neighbors/adjacent to c. A straightforward algorithm to get an interaction list is to follow the definition. From the Morton index m_c of a cell c, we obtain c's parent index with a bit shift $m_p = \mathit{bits\_shift\_right}(m_c, 3)$. To find the neighbors of c's parent p, we convert m_p in grid-coordinate (x_p, y_p, z_p) using the Algorithm 19. The grid-coordinate of p's neighbors are in the intervals $(x_p \pm 1, y_p \pm 1, z_p \pm 1)$. For each neighbor g, we generate its Morton index m_g using Algorithm 18 and obtain its children indexes with a bit shift. We test the children indexes to ensure that no direct neighbors of c are added to the list. Two cells are

Algorithm 18: Grid-Coordinate to Morton index

Data: C_m the position of the cell, l the level of the cell
function `GetMortonIndex(`C_m`, l) : mindex`
    mindex = 0;
    mask = 1;
    `// The order is xyz.xyz....`
    C_m[0] = bits_shift_left(2, C_m[0]);
    C_m[1] = bits_shift_left(1, C_m[1]);
    `//` C_m`[2] is unchanged`
    for `indexLevel = 0` $\rightarrow$ `l` **do**
        mindex = mindex OR (C_m[2] AND mask);
        mask = bits_shift_left(mask, 1);
        mindex = mindex OR (C_m[1] AND mask);
        mask = bits_shift_left(mask, 1);
        mindex = mindex OR (C_m[0] AND mask);
        mask = bits_shift_left(mask, 1);
        C_m[2] = bits_shift_left(C_m[2], 2);
        C_m[1] = bits_shift_left(C_m[1], 2);
        C_m[0] = bits_shift_left(C_m[0], 2);
    end
    return mindex;

Algorithm 19: Morton index to grid-coordinate

Data: *mindex* the Morton index of the cell, l the level of the cell
function `GetTreeCoordinate(mindex, l) :` C_m
    mask = 1;
    C_m[:] = 0;
    `// For all levels from root to the cell level`
    for `indexLevel = 0` $\rightarrow$ `l` **do**
        `// z position`
        C_m[2] = C_m[2] OR (mask AND mindex);
        mindex = bits_shift_right(m, 1);
        `// y position`
        C_m[1] = C_m[1] OR (mask AND mindex);
        mindex = bits_shift_right(m, 1);
        `// x position`
        C_m[0] = C_m[0] OR (mask AND mindex);
        mask = bits_shift_left(mask, 1);
    end
    return C_m ;

not direct neighbors if they are separated by more than one cell in at least one direction. The drawback of this method is the conversion from Morton index to grid-coordinate and then again from grid-coordinate into Morton index.

An alternative is to stay in the Morton index side and to work with binary operations. For example, we test the if two cells are neighbors using their Morton indexes with the Algorithm 20. The complexity of this test is the working dimension and thus is it constant in our case. The idea is to look if there is one bit difference between two indexes. If we are in a $1D$ problem, two indexes A and B (with $A > B$) are neighbors if there are equals, if only the lower bit is different or if $A = B + 1$. However, in higher dimensions, the bits are interleaved such that we cannot use the basic arithmetic operations ($+/-/\times/\div$). Therefore, to test $A = B+1$ we use binary operations:

first we find the last non-zero bit position in a p, then A and B should be equal for all the bits above p and B should be one for the bits before p.

Algorithm 20: Test to know if two Morton indexes are neighbors

```
Data: mindex_A the first index, mindex_B the second index
function AreNeighbors(mindex_A, mindex_B) : true if indexes are neighbors
    mask_x = 100.100...100.100 bits;
    mask_y = 010.010...010.010 bits;
    mask_z = 001.001...001.001 bits;
    flags[3] = mask_x, mask_y, mask_z ;
    for idxDim = 0 → 3 do
        vA = (mindex_A AND flags[idx]);
        vB = (mindex_B AND flags[idx]);
        // Test if equal or if only the last bit is different
        if (vA == vB) or ((vA XOR vB) == 1) then
            // Are neighbor...
        else
            // bsf instruction
            firstBit = first_nnz_bit(Max(vA, vB));
            highMask = bits_shift_left(bits_shift_right(NOT(0), firstBit+1) , firstBit+1) AND flags[idx];
            if (vA AND highMask) ≠ (vB AND highMask) then
                return false;
            end
            lowMask = (NOT bits_shift_left((NOT(0)) , firstBit)) AND flags[idx];
            if (Min(vA,vB) AND lowMask) ≠ lowMask then
                return false;
            end
            // Are neighbor...
        end
    end
    return true;
```

The Algorithm 21 returns what we call the directive indexes which are composed of 3 Morton indexes; the first index is the Morton coordinate of the cells at the relative position $(-1,-1,-1)$, the second index is equal to the tested Morton index m and the last index is the Morton index of the of the cells at the relative position $(+1,+1,+1)$. From these relative indexes we can generate the direct neighbors of m (the cells ± 1 in each direction) with Algorithm 22. Using these algorithms, we are able to get the Morton index of the neighbors of a cell without generating the grid-coordinates.

D.3 Quicksort in Distributed Memory

The distribution of the data before the construction of the octree is a critical stage to ensure a scalable FMM. The distributed Quicksort, taken from [118], is a great algorithm to sort the particles over a large number of MPI processes. Despite its ambiguous name, the distributed Quicksort is not composed by local Quicksort and the data are not locally sorted. The algorithm is similar to a local Quicksort but it stops once there are enough recursions to ensure one partition per process as presented in Algorithm 23. As the usual Quicksort, the choice of the pivot is crucial to ensure performance but also to guarantee that all the data will not finally be hosted by a single node.

The choice of the pivot is done globally and is used by all the processes from a group to partition

Algorithm 21: Morton index to directive indexes

```
Data: mindex the Morton index of the cell
function GetDirectiveIndexes(mindex) : directiveIndexes[3]
    mask_x = 100.100...100.100 bits;
    mask_y = 010.010...010.010 bits;
    mask_z = 001.001...001.001 bits;
    // The center index is equal to mindex
    directiveIndexes[1] = mindex;
    // Compute the negative index
    mindex_minus = mindex;
    flag_minus = 0;
    if (((mindex AND mask_x) XOR mask_x) ≠ mask_x then
        flag_minus = flag_minus OR 4;
    end
    if (((mindex AND mask_y) XOR mask_y) ≠ mask_y then
        flag_minus = flag_minus OR 2;
    end
    if (((mindex AND mask_z) XOR mask_z) ≠ mask_z then
        flag_minus = flag_minus OR 1;
    end
    while flag_minus is not 0 do
        prevflag_minus = flag_minus ;
        flag_minus = bits_shift_left(flag_minus AND NOT(mindex_minus), 3) ;
        mindex_minus = (mindex_minus XOR prevflag_minus) ;
    end
    directiveIndexes[0] = mindex_minus;
    // Compute the positive index
    mindex_plus = mindex; flag_plus = 0;
    if (mindex AND limiteX) == limiteX then
        flag_plus = flag_plus OR 4;
    end
    if (mindex AND limiteY) == limiteY then
        flag_plus = flag_plus OR 2;
    end
    if (mindex AND limiteZ) == limiteZ then
        flag_plus = flag_plus OR 1;
    end
    while flag_plus ≠ 0 do
        prevflag_plus = flag_plus;
        flag_plus = bits_shift_left(flag_plus AND mindex_plus , 3);
        mindex_plus = mindex_plus XOR prevflag_plus;
    end
    directiveIndexes[2] = mindex_plus;
```

Algorithm 22: Morton index of the neighbor of m from the directive indexes

```
Data: directiveIndexes[3] the result of GetDirectiveIndexes
directions[3] +1/ − 1 in each dimension
function GetNeighborIndex(directiveIndexes[3], directions[3]) : neighborIndex
    mask_x = 100.100...100.100 bits;
    mask_y = 010.010...010.010 bits;
    mask_z = 001.001...001.001 bits;
    return directiveIndexes[directions[0]+1] AND mask_x
    OR directiveIndexes[directions[1]+1] AND mask_y
    OR directiveIndexes[directions[2]+1] AND mask_z;
```

their data. Then, the first half of the processes will handle the lowest values whereas the other half will manage the greater values. After a communication stage which makes the processes exchanging their lower or upper partition, the process group is divided appropriately to repeat the operation. Once a process is alone in its group it sorts the data locally with any sorting algorithm.

To have an optimized implementation, the division of the group should be done proportionally to the number of elements lower/greater than the pivot. The choice of the pivot must include the cases where some processes do not have values or all processes have the same values.

Algorithm 23: Distributed Quicksort

```
Data: values an array of values to sort, N the size of the array
mpi_group the group of process that hold data.
Output: sorted values not lower than the values hosted by lower rank processes and not greater than the values
hosted by higher rank processes.
function DistributedQS(values, N, mpi_group) : values
    while number of process in mpi_group ≠ 1 do
        pivot = SelectPivot(values, mpi_group);
        [values, part] = Partition(values, pivot);
        totalLowerPart[mpi_group.size] = ReduceSum(mpi_group, part);
        totalGreaterPart[mpi_group.size] = ReduceSum(mpi_group, N-part);
        if current process belong to lower part then
            destRank = SelectProcess(mpi_group, totalLowerPart, totalGreaterPart);
            Send(destRank, values[part:end]);
            values = [ values[0:part] : Recv(mpi_group)];
            mpi_group = ReduceGroup(mpi_group);
        else
            destRank = SelectProcess(mpi_group, totalLowerPart, totalGreaterPart);
            Send(destRank, values[part:end]);
            values = [ values[0:part] : Recv(mpi_group)];
            mpi_group = ReduceGroup(mpi_group);
        end
    end
    local_sort(values);
    return values;
```

D.4 Generic Periodicity Algorithm for the FMM

Several physical problems require periodic boundary conditions and numerous specific methods have been developed to simulate the repetition of the simulation box. An illustration of the periodicity is shown by Figure D.2. In this section, we present an algorithm which uses the usual FMM operators to generate the periodicity and which has the same accuracy as the unerlying FMM kernel. The extra number of FMM operations to perform is low and it generates a large number of repetitions of the simulation box in one or several directions/dimensions. In a classic FMM, the far-field stops at level 2 (at level 1 there are 8 neighbor cells and thus no possible $M2L$) whereas the key idea of our approach is to continue the FMM above the level 2.

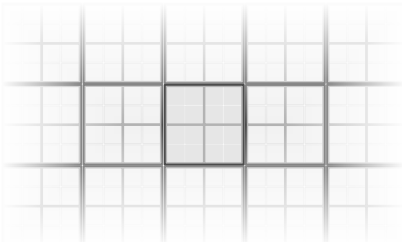


Figure D.2: Repetition of the simulation box in $2D$ in all directions

D.4.1 Periodicity up to the Level 2

The periodic boundary condition is applied by the $P2P$ and the $M2L$ operators by simulating a periodicity when we generate the neighbor/interaction lists. At level l there are $g = 2^l$ cells in each dimension, and if we ask for a cell outside the grid we round to find the corresponding cell inside the grid if it exists; if the index is < 0 we add g and if if is $>= g$ we reduce by g. This step repeats the simulation box by half in all the directions.

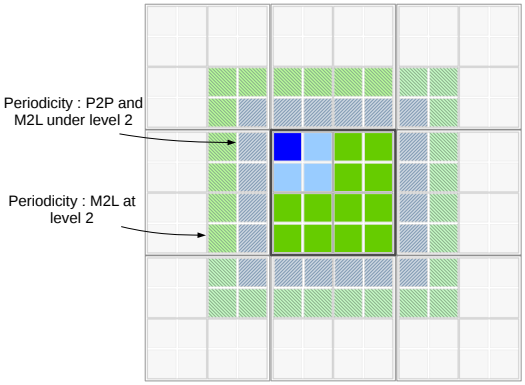


Figure D.3: Repetition of the simulation box until level 2. The box is repeated 0.5 time in each dimension. The dense blue cell of level 2 covers the dark green cells.

D.4.2 Periodicity up to the Level 1

The same principle can be applied by continuing the FMM up to the level 1 even if the usual FMM stops at level 2. In this case, it simulates a repetition of one box in each dimension. The order of the operations is standard to the FMM, the upward pass should go up to the level 1 and the $L2L$ $1 \rightarrow 2$ should be done after the $M2L$ at level 1. The interaction list of a cell c at level 1 includes c itself and has duplicate elements but this should not be a problem because it simply means that the same data distribution exists like if the elements were replicated.

D.4.3 Periodicity up to the root (level 0)

We apply the FMM up to the root level. The root cell does not have any parent and therefore it does not have any child position which is used to compute the interaction list of the $M2L$. The classic interaction list includes the cells in the relative interval $-3/2$ or $-2/3$ depending on the position of the target relatively to its parent. In order to have a large repetition while we still use

the usual $M2L$ we decide that the interaction list of the root includes the cells from the interval $-3/3$. The interactions list of the root is composed by itself and is made of 316 elements instead of 189 in the usual $M2L$. This repetition is shown in Figure D.4 and we see that the real simulation box is repeated 3 times in each direction.

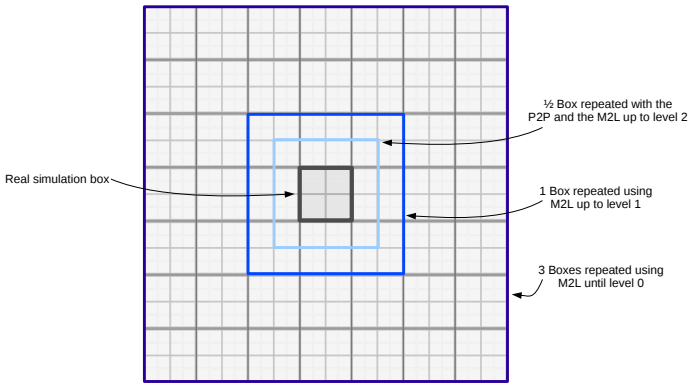


Figure D.4: Repetition of the simulation box until level 0. The box is repeated 3 times in each dimensions and each directions.

D.4.4 Periodicity Above the Root (Above the Level 0)

The periodic FMM algorithm continues above our real octree to have larger repetitions. It is straightforward to create the cell above the root by simply considering that our simulation box is repeated. The cell of level -1 (one level above the root) is obtained by a single $M2M$ where the 8 children are all the root cell. The idea is valid until any level by simply considering that the 8 children are equal to the lower-level cell. However, some important choices should be made to perform the $M2L$ and to choose where our real octree is located in the virtual larger octree.

Our objective is to put our real octree in the center of the virtual grid generated by going above the root and thus we try to have almost the same number of repetitions in all the directions. Moreover, to ensure a correct coherency and to avoid some duplicate the $M2L$ at level 0 should not be done in $-3/3$ if we continue the algorithm above the root. First, we consider that the real root is the last child relatively to its parent $(+1, +1, +1)$. For all the upper levels, we consider that the parents of the real root are the first child of their respective parent $(0, 0, 0)$. Therefore, the $M2L$ at level 0 is done on the interval $-3/2$, and for the upper levels in the interval $-2/3$. By doing so, we still used the usual FMM operators and we repeat the original simulation box in all directions as shown in Figure D.5 but there is a difference of 1 in each dimension. This single difference becomes negligible as the repetitions get larger.

This strategy is applied without any limit apart the numerical stability of the underlying kernel. When the periodicity algorithm is applied up to the $-l$ levels above the root we generate a grid of $6 \times 2^{|l|}$ times the original box; each $M2M$ above the root doubles the grid dimension such that $-l$ levels above the root a cell represents $2^{|l|}$ times the original box. Moreover, at the last level the $M2L$ is done from -2 to 3 and covers 6 cells. In $3D$, the total repetition of the box in the simulation is $B = (6 \times 2^{|l|})^3$, and we have for $l = 1$ $B = 1728$, for $l = 5$ $B = 7,077,888$ and

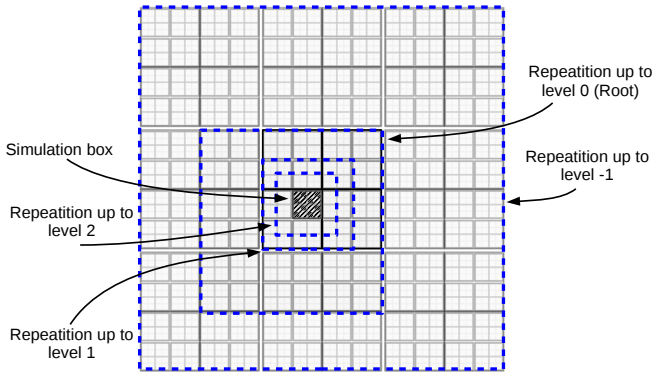


Figure D.5: Repetition of the simulation box until level -1. The box is repeated -5 and $+6$ times in all dimensions.

for $l = 10$ $B = 231 \cdot 10^9$. Regarding the number of operations, for each level above the root we compute one $M2M$ with 8 children (a complete $M2M$), a complete $M2L$ with 189 interactions and one $L2L$ with a single child. To prepare a normal FMM kernel for a periodic computation, we have to specify that we compute a FMM with a tree of height $h + |l| + 2$ and that the simulation box is of width $w * 2^{|l|+2}$, with w the original width. Therefore, we choose the l parameters depending on the number of repetitions we want to guarantee enough periodicity.

D.5 Heteroprio Scheduler

During our researches on the runtime based FMM, we developed a scheduler to manage the heterogeneous architectures. This scheduler called Heteroprio allow to assign one priority per processing unit type to each task. We present here the Heteroprio implementation using some StarPU terminology. A schematic view of the scheduler is presented in Figure D.6.

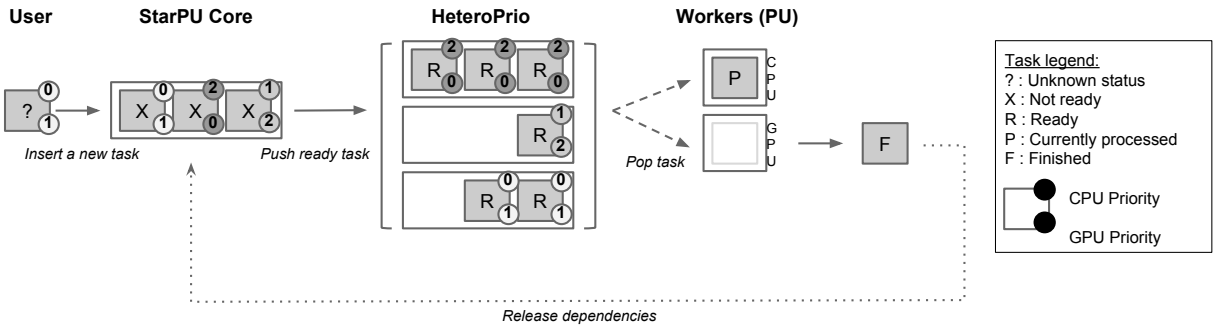


Figure D.6: Heteroprio scheduler.

The implementation of Heteroprio is straightforward if the priorities are static n-tuple with n the number of processing unit types. We create one bucket per n-tuple and we define which kind of processing unit is authorized to pick a task from this bucket. Then, for each type of processing unit we create an indirection access to the different buckets from the lower to higher priorities.

As an example, if we have four buckets: $B0 : \{CPU, GPU\}$, $B1 : \{CPU, GPU, OPENCL\}$, $B2 : \{CPU, GPU\}$ and $B3 : \{CPU\}$. At least one processing unit type must point to each bucket otherwise the tasks inserted into a bucket will not be computed. But it is not necessary for all

the architectures enabled in a bucket to point on it. We can have the following priorities; $CPU : \{B0, B1, B3\}$, $GPU : \{B2, B0\}$ and $OPENCL : \{B1\}$. A priority list defines where a worker will look and in which order, such that in our example for a CPU the tasks in $B0$ are more prioritized than the tasks in $B1$. Of course, it is also possible to add a prefetch system but doing so reduces the visibility of the priorities.

Heteroprio also includes a management of the *critical* state. For each of the above buckets, the user is able to specify which type is the fastest processing unit. Then, for each other type of processing unit which can access this bucket, the user specifies its $slow - factor$ which tells how slow are the other types against the fastest one. Then, Heteroprio gives a task to a worker only if it does not result in a possible time extension. As an example, if we have a bucket with two processing unit enabled $b : P^0, P^1$. If P^0 is the faster type and we have a slow factor $P^1/P^O = F^1$. Then Heteroprio gives a task from b to a worker of type P^0 only if there are more than $F^1 \times N^0$ tasks, where N^0 is the number of worker of type P^0. For example, if $P^1/P^O = 3$ and $N^0 = 2$, a worker of type P^1 gets a task from b if there are more than 6 tasks. It might not give the optimal execution, but if the tasks in b are the last ones, it reduce the wrong choice possibilities by giving a task to a slow worker which finally extend the wall duration.

E

HPC Software Engineering

In this Appendix, we discuss different paradigms and methods related to software engineering and HPC.

E.1 *C++* Language

The different libraries developed during this thesis have been done in *C++*. The *C++* is an object-oriented programming language created in the 1980 decade by Bjarne Stroustrup. Since its birth, the *C++* has been upgraded by several standards with the most recent in 2014. Nowadays, it is still in the tops 10 of the most-used languages and it is the main language for many large projects. Some illustrative examples are Gcc, LLVM, Adobe products, several server applications (Amazon, Amadeus, Facebook, Google, ...), several Microsoft softwares and some parts of operating systems (Apple, Microsoft). In the HPC community, the *C* or the *Fortran* are widely used for historical reasons but the *C++* seems more and more popular. The *Fortran* is convenient to write linear accesses and multi-dimensional matrices which is appreciated by the applied-mathematics community especially in linear algebra. The power of the object-oriented paradigm may result in a dramatic design but, if used appropriately, it is a clear asset for maintainability and large projects' life-cycle. Moreover, the *C++* keeps some compatibilities with the *C* and it is possible to link *C++*, *C* and *Fortran* binaries. Therefore, it is possible to develop or to use old kernels in *C*/*Fortran* inside a *C++* project. In addition, *C++* is a compiled language which is a required property in HPC.

E.1.1 *C++* Template and Header Template Library (HTL)

Among the various functionalities and features of the *C++*, the *templates* are a great help to write optimized code while keeping some genericity as discussed in Appendix E.2. The templates are categorized in three parts: the function templates, the class templates and the variable templates. The function/class/type which is used in place of the template is known at compile-time which opens the possibilities for several optimizations by the compiler; the compiler is able to apply specific and low level optimizations because it knows what it works with. For example, knowing a function at compile allows to *inline* the target function and potentially to avoid aliasing. While the *C++* virtual methods help for a good design, they may results in the use of function pointers evaluated at runtime.

The resulting libraries of our study are based on the coding pattern *Header Template Library* which itself is composed by the *all-in-header* pattern and then *pure-template* pattern. All the *C++* classes are developed inside their header files and majority of them are template classes. While it is not usual and maybe not appropriate for huge projects, it has some nice assets. On the other hand, the main drawback of the HTL pattern is the compilation time because no intermediate object files are generated and when an executable is compiled all the included headers are re-compiled. However, in HPC we concentrate on the execution time and we agree to pay an extra-cost during the compilation stage if it is beneficial for the execution. Building a library with the Header Template Library pushes the compiler to perform function inlining and to know what the called functions do. We can pre-compile specialized template classes to accelerate the compilation (especially for template classes which are widely used). We can still use virtual functions at high level to create an appropriate design but the HTL pattern removes the virtual functions and the use of interfaces which improves the performance at low level in a computational loop for example. The HTL pattern limits the circular dependencies and uses compiled time polymorphism which helps to have a maintainable design. However, some expressions are difficult to express in template especially if it generates in a variable number of variables. The straightforward method is to use an array to correlate a template parameter but it might not result in the same binary as if we write the function with different variables. For example, the declaration of $int\ A[S];$, with S a template might not give the same binary as $int\ a,\ b;$ even if $S = 2$; In addition, the source of the HTL libraries is distributed with the library header and if one wants to hide the source code, he must use an interface and provide only pre-compiled parts of the template classes.

E.2 Genericity vs. Optimization

During the development of an HPC application, the question of keeping an algorithm generic or applying optimizations is asked numerous times. HPC is about optimization and reducing the execution time but the design should not be neglected in a long-term project. Software engineering is about design, maintainability, modularity and re-usability where the genericity occupies a special place. Therefore, they appear opposed and while an algorithm is more and more optimized for a given input or a given hardware it looses its genericity. The question must be asked whether if

we agree to lose this genericity to have a potential gain in performance. Of course, it is always possible to maintain different versions of an algorithm with distinct degrees of optimizations but this is acceptable only for extremely mature kernels which will not be updated in the near future.

The *C++* templates reduce the gap and allow to have generic algorithm and potentially a high degree of optimizations. In our libraries, some kernels are completely generic and are *templatized* with scalar or SIMD data type in double or single precision. But some optimizations are up-to the compilers and there exist some special cases where writing the optimized code is better than a generic template-optimized code.

The genericity of an application is also tied to the hardware; developing a kernel for a special type of processing unit or even tuning an application for a given machine. This is not a viable position, and any application should be easily upgraded to a new hardware. For example, we cannot be enthusiastic about an application which works only with a given number of processors or which needs to be rewritten if we have more than x GPUs. That is why it is preferable to develop on top of a runtime system to delegate a part of the hardware management. Runtime systems specialized for heterogeneous architectures (as StarPU) help a lot to have an abstraction of the hardware and to define the core algorithm of an application independently from the hardware. The initial development cost is expensive, but the gain in maintainability and modularity is huge.

E.3 OpenCL Meta-Programming

OpenCL is a framework dedicated to the execution of applications across heterogeneous architectures. It includes a language standard (based on the C99) to write programs which can be compiled for different devices which propose the appropriate driver. It is common to compile the OpenCL code at runtime by asking the driver of the target device to transform the source code into an executable binary. In the research, OpenCL is not as used as CUDA and if we look at the number of results for the queries *OpenCL* and *CUDA* we get 33 and 86 items in Elsevier Parallel Computing journal, and $19,500$ and $80,300$ results in Google Scholar. The fact that CUDA is a compiled language and match the NVidia GPU specificities makes it more appropriate to perform advanced optimizations. While the main objective of OpenCL is portability, the performance of a kernel are not portable. However, compilation at runtime offers impressive capabilities, because we can transform the source during the execution and compiled a new binary. Therefore, it gives the possibility for our programs to generate new programs which is related to the concept of meta-programming but also tuning at runtime.

Using the compilation at runtime, we are able to do what is supported by the *C++* template but also much more advanced code generation. As an example, the Code E.1 compute a SpMV with a row blocking pattern. This function works for the *double* data type and any length of block (*block_size*). If we want the same function of any other data type, we must duplicate the code. In addition, several optimization cannot be done because the length of the block is evaluate at runtime when the code is executed on the device.

```
1 __kernel void spmv_c(__global const double* __restrict matrix_values,
2                       __global const int* __restrict matrix_i,
```

```
            __global const int* __restrict matrix_j,
            const int matrix_n,
            const int block_size,
            __global const double* __restrict vec_a,
            __global double* __restrict vec_b){
    int idxValue, idxBlock;
    for(idxValue = 0 ; idxValue < matrix_n ; ++idxValue){
        double sum = 0;
        __global const double* ptrValues = &matrix_values[idxValue];
        __global const double* ptrA      = &vec_a[matrix_j[idxValue]];
        for(idxBlock = 0 ; idxBlock < block_size ; ++idxBlock){
            sum += ptrA[idxBlock] * ptrValues[idxBlock];
        }
        vec_b[matrix_i[idxValue]] += sum;
    }
}

```

Code E.1: SpMV with row blocking in OpenCL

In Code E.2 we give the same function but with the use of meta-programming. At runtime, the file is loaded but before to compiled it we must replace the keywords *@TYPE* and *@SIZE* by appropriate values. For example, if we find during an analysis that the single floating point values are enough accurate and that a block length of 7 matchs perfectly our problem, we replace the text and create a function *spmv_float_7*. Executions of this SpMV example on an $Intel\ i7-4610M$ CPU at $3.00GHz$ show that the meta-code (E.2) is 2 times faster than the classic code (E.1).

```
__kernel void spmv_@TYPE_@SIZE(__global const @TYPE* __restrict matrix_values,
                    __global const int* __restrict matrix_i,
                    __global const int* __restrict matrix_j,
                    const int matrix_n,
                    __global const @TYPE* __restrict vec_a,
                    __global @TYPE* __restrict vec_b){
    int idxValue, idxBlock;
    for(idxValue = 0 ; idxValue < matrix_n ; ++idxValue){
        @TYPE sum = 0;
        __global const @TYPE* ptrValues = &matrix_values[idxValue];
        __global const @TYPE* ptrA      = &vec_a[matrix_j[idxValue]];
        for(idxBlock = 0 ; idxBlock < @SIZE ; ++idxBlock){
            sum += ptrA[idxBlock] * ptrValues[idxBlock];
        }
        vec_b[matrix_i[idxValue]] += sum;
    }
}

```

Code E.2: Meta-SpMV with row blocking in OpenCL

We can replace or remove large portions of the code to take into account the target device specificities but also to give more information to the compiler and to create a kernel which is well suited for our problem. Using *C++* template to do the same thing requires to compile all the possibilities which is expensive and increases the size of the final binary. Among the high level optimizations, we can decide to use a shared memory buffer if the target hardware does not have caches or we can parametrize the size of some arrays based on the hardware cache sizes. At a lower level, we can set values to help the compiler, unroll loop (which include duplicate source

line and unroll by-hand), avoid indirections if we have a repeated pattern or evaluate expressions. Therefore, by building a modular code made of several pieces and customizing at runtime our source, we may succeed to have portable performance.

E.4 Memory Allocations

The memory allocations are critical in the development of HPC applications such that the computational parts of an application must not allocate any data dynamically. All the dynamic allocations should be done during an initialization stage. Moreover, in *C++* one should use the external libraries carefully and for example, it looks inappropriate to declare a standard vector inside a kernel since it generates dynamic allocations and complex behaviors.

Additionally, the memory should be allocated in one shot because re-allocation generally gives poor performances (even in the initialization stage) since it generates one allocate and one copy at each call. It is usually better to find out the size of the final block in a first stage, allocate the entire block and fill it in a second stage even if the block size estimation and the block filling are almost similar. The data locality should be enhanced by avoiding the use of pointers in the attributes of the *C++* class or in linked lists of small nodes. In *C++* the computational data classes should be POD (Plain Old Data) and allocated by block.

Listing of figures

List of Tables

List of Algorithms

List of Codes

References

[1] YJ Liu, S Mukherjee, N Nishimura, M Schanz, W Ye, A Sutradhar, E Pan, NA Dumont, A Frangi, and A Saez. Recent advances and emerging applications of the boundary element method. *Applied Mechanics Reviews*, 64(3):030802, 2011.

[2] Isabelle Terrasse. *Résolution mathématique et numérique des équations de Maxwell instationnaires par une méthode de potentiels retardés*. PhD thesis, 1993.

[3] Guillaume Sylvand. Équation des Ondes en Acoustique : Accélération des Potentiels Retardés par la Méthode Multipôle Temporelle. Technical Report RR-5017, INRIA, November 2003.

[4] Gernot Beer, Ian Smith, and Christian Duenser. *The boundary element method with programming: for engineers and scientists*. Springer Science & Business Media, 2008.

[5] Top 500 supercomputer sites, May 2015.

[6] OpenMP Specifications. Openmp application programing interface. v3. 0, may 2008, 2012.

[7] M Snir, S Otto, S Huss-Lederman, D Walker, and J Dongarra. Mpi–the complete reference, 2nd edn. the mpi core, vol. 1, 1998.

[8] Khronos Group. *The OpenCL Specification*, September 2010.

[9] *CUDA Programming Guide 7.0*. NVIDIA, 2015.

[10] Emmanuel Agullo, Jim Demmel, Jack Dongarra, Bilel Hadri, Jakub Kurzak, Julien Langou, Hatem Ltaief, Piotr Luszczek, and Stanimire Tomov. Numerical linear algebra on emerging architectures: The plasma and magma projects. In *Journal of Physics: Conference Series*, volume 180, page 012037. IOP Publishing, 2009.

[11] Field G Van Zee, Ernie Chan, Robert A Van de Geijn, Enrique S Quintana-Ort, and Gregorio Quintana-Ort. The libflame library for dense matrix computations. *Computing in science & engineering*, 11(6):56–63, 2009.

[12] Xavier Lacoste, Mathieu Faverge, George Bosilca, Pierre Ramet, and Samuel Thibault. Taking advantage of hybrid systems for sparse direct solvers via task-based runtimes. In *Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, pages 29–38. IEEE, 2014.

[13] Emmanuel Agullo, Alfredo Buttari, Abdou Guermouche, and Florent Lopez. Multifrontal qr factorization for multicore architectures over runtime systems. In *Euro-Par 2013 Parallel Processing*, pages 521–532. Springer, 2013.

[14] Alin Murarașu, Josef Weidendorfer, and Arndt Bode. Workload balancing on heterogeneous systems: A case study of sparse grid interpolation. In *Euro-Par 2011: Parallel Processing Workshops*, pages 345–354. Springer, 2012.

[15] Canqun Yang, Feng Wang, Yunfei Du, Juan Chen, Jie Liu, Huizhan Yi, and Kai Lu. Adaptive optimization for petascale heterogeneous cpu/gpu computing. In *Cluster Computing (CLUSTER), 2010 IEEE International Conference on*, pages 19–28. IEEE, 2010.

[16] George Em Karniadakis and Robert M Kirby II. *Parallel scientific computing in C++ and MPI: a seamless approach to parallel algorithms and their implementation*, volume 1. Cambridge University Press, 2003.

[17] James Hardy Wilkinson, C Reinsch, Friedrich L Bauer, et al. *Handbook for automatic computation: linear algebra*, volume 2. Springer-Verlag New York, 1971.

[18] Richard Wilson Vuduc. *Automatic performance tuning of sparse matrix kernels*. PhD thesis, Citeseer, 2003.

[19] Eun-Jin Im. *Optimizing the Performance of Sparse Matrix-Vector Multiplication*. PhD thesis, EECS Department, University of California, Berkeley, Jun 2000.

[20] James B White III and P Sadayappan. On improving the performance of sparse matrix-vector multiplication. In *High-Performance Computing, 1997. Proceedings. Fourth International Conference on*, pages 66–71. IEEE, 1997.

[21] Sivan Toledo. Improving the memory-system performance of sparse-matrix vector multiplication. *IBM Journal of research and development*, 41(6):711–725, 1997.

[22] Ali Pinar and Michael T Heath. Improving performance of sparse matrix-vector multiplication. In *Proceedings of the 1999 ACM/IEEE conference on Supercomputing*, page 30. ACM, 1999.

[23] Eun-Jin Im, Katherine Yelick, and Richard Vuduc. Sparsity: Optimization framework for sparse matrix kernels. *International Journal of High Performance Computing Applications*, 18(1):135–158, 2004.

[24] Richard W Vuduc and Hyun-Jin Moon. Fast sparse matrix-vector multiplication by exploiting variable block structure. In *High Performance Computing and Communications*, pages 807–816. Springer, 2005.

[25] Yousef Saad. Sparskit: a basic tool kit for sparse matrix computations, 1994.

[26] Yousef Saad. Iterative methods for sparse linear systems second edition. *Philadelphia, Pennsylvania: SIAM*, 2003.

[27] Richard Vuduc, James W Demmel, and Katherine A Yelick. Oski: A library of automatically tuned sparse matrix kernels. In *Journal of Physics: Conference Series*, volume 16, page 521. IOP Publishing, 2005.

[28] Eun-Jin Im and Katherine Yelick. Optimizing sparse matrix computations for register reuse in sparsity. In *Computational Science–ICCS 2001*, pages 127–136. Springer, 2001.

[29] Elizabeth Cuthill and James McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 1969 24th national conference*, pages 157–172. ACM, 1969.

[30] Juan Carlos Pichel, Dora Blanco Heras, José Carlos Cabaleiro, and Francisco F Rivera. Performance optimization of irregular codes based on the combination of reordering and blocking techniques. *Parallel Computing*, 31(8):858–876, 2005.

[31] Michele Martone, Salvatore Filippone, Salvatore Tucci, Marcin Paprzycki, and Maria Ganzha. Utilizing recursive storage in sparse matrix-vector multiplication-preliminary considerations. In *CATA*, pages 300–305, 2010.

[32] AN Yzelman. Fast sparse matrix-vector multiplication by partitioning and reordering. 2011.

[33] Gundolf Haase, Manfred Liebmann, and Gernot Plank. A hilbert-order multiplication scheme for unstructured sparse matrices. *International Journal of Parallel, Emergent and Distributed Systems*, 22(4):213–220, 2007.

[34] Samuel Williams, Leonid Oliker, Richard Vuduc, John Shalf, Katherine Yelick, and James Demmel. Optimization of sparse matrix-vector multiplication on emerging multicore platforms. *Parallel Computing*, 35(3):178–194, 2009.

[35] Juan Carlos Pichel, Dora Blanco Heras, José Carlos Cabaleiro, and Francisco F Rivera. Increasing data reuse of sparse algebra codes on simultaneous multithreading architectures. *Concurrency and Computation: Practice and Experience*, 21(15):1838–1856, 2009.

[36] Brendan Vastenhouw and Rob H Bisseling. A two-dimensional data distribution method for parallel sparse matrix-vector multiplication. *SIAM review*, 47(1):67–95, 2005.

[37] Aydin Buluç, Jeremy T Fineman, Matteo Frigo, John R Gilbert, and Charles E Leiserson. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, pages 233–244. ACM, 2009.

[38] Vasileios Karakasis, Georgios Goumas, and Nectarios Koziris. A comparative study of blocking storage methods for sparse matrices on multicore architectures. In *Computational*

Science and Engineering, 2009. CSE'09. International Conference on, volume 1, pages 247–256. IEEE, 2009.

[39] Muthu Manikandan Baskaran and Rajesh Bordawekar. Optimizing sparse matrix-vector multiplication on gpus using compile-time and run-time strategies. *IBM Reserach Report, RC24704 (W0812-047)*, 2008.

[40] Michael Garland. Sparse matrix computations on manycore gpu's. In *Proceedings of the 45th annual Design Automation Conference*, pages 2–6. ACM, 2008.

[41] Nathan Bell and Michael Garland. Efficient sparse matrix-vector multiplication on cuda. Technical report, Nvidia Technical Report NVR-2008-004, Nvidia Corporation, 2008.

[42] Nathan Bell and Michael Garland. Implementing sparse matrix-vector multiplication on throughput-oriented processors. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, page 18. ACM, 2009.

[43] Alexander Monakov, Anton Lokhmotov, and Arutyun Avetisyan. Automatically tuning sparse matrix-vector multiplication for gpu architectures. In *High Performance Embedded Architectures and Compilers*, pages 111–125. Springer, 2010.

[44] Dominik Grewe and Anton Lokhmotov. Automatically generating and tuning gpu code for sparse matrix-vector multiplication from a high-level representation. In *Proceedings of the Fourth Workshop on General Purpose Processing on Graphics Processing Units*, page 12. ACM, 2011.

[45] Richard Vuduc, Aparna Chandramowlishwaran, Jee Choi, Murat Guney, and Aashay Shringarpure. On the limits of gpu acceleration. In *Proceedings of the 2nd USENIX conference on Hot topics in parallelism*, pages 13–13. USENIX Association, 2010.

[46] Chetan Jhurani and Paul Mullowney. A gemm interface and implementation on nvidia gpus for multiple small matrices. *Journal of Parallel and Distributed Computing*, 75:133–140, 2015.

[47] *Intel Math Kernel Library. Reference Manual*. Intel Corporation, 2009.

[48] Vasileios Karakasis, Georgios Goumas, and Nectarios Koziris. Perfomance models for blocked sparse matrix-vector multiplication kernels. In *Parallel Processing, 2009. ICPP'09. International Conference on*, pages 356–364. IEEE, 2009.

[49] Rajesh Nishtala, Richard W Vuduc, James W Demmel, and Katherine A Yelick. When cache blocking of sparse matrix vector multiply works and why. *Applicable Algebra in Engineering, Communication and Computing*, 18(3):297–311, 2007.

[50] Dahai Guo and William Gropp. Applications of the streamed storage format for sparse matrix operations. *International Journal of High Performance Computing Applications*, 28(1):3–12, 2014.

[51] A. Duran, J. M. Perez, R. M. Ayguadé, E. amd Badia, and J. Labarta. Extending the OpenMP tasking model to allow dependent tasks. In *OpenMP in a New Era of Parallelism, 4th International Workshop, IWOMP 2008*, West Lafayette, IN, May 12-14 2008. Lecture Notes in Computer Science 5004:111-122. DOI: 10.1007/978-3-540-79561-2_10.

[52] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009*, 23:187–198, February 2011.

[53] George Bosilca, Aurélien Bouteiller, Anthony Danalis, Mathieu Faverge, Thomas Hérault, and Jack Dongarra. PaRSEC: A programming paradigm exploiting heterogeneity for enhancing scalability. *Computing in Science and Engineering*, 99:1, 2013.

[54] Zoran Budimlić, Michael Burke, Vincent Cavé, Kathleen Knobe, Geoff Lowney, Ryan Newton, Jens Palsberg, David Peixotto, Vivek Sarkar, Frank Schlimbach, and Sağnak Taşirlar. Concurrent collections. *Sci. Program.*, 18(3-4):203–217, August 2010.

[55] A. YarKhan, J. Kurzak, and J. Dongarra. QUARK users' guide: QUeueing And Runtime for Kernels. Technical Report ICL-UT-11-02, Innovative Computing Laboratory, University of Tennessee, April 2011. `http://icl.cs.utk.edu/projectsfiles/plasma/pubs/56-quark_users_guide.pdf`.

[56] E. Chan, E. S. Quintana-Orti, G. Gregorio Quintana-Orti, and R. van de Geijn. Supermatrix Out-of-Order Scheduling of Matrix Operations for SMP and Multi-Core Architectures. In *Nineteenth Annual ACM Symposium on Parallel Algorithms and Architectures SPAA'07*, pages 116–125, June 2007.

[57] Gregorio Quintana-Orti, Enrique S. Quintana-Orti, Ernie Chan, Robert A. van de Geijn, and Field G. Van Zee. Scheduling of qr factorization algorithms on smp and multi-core architectures. In *Proceedings of the 16th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP 2008)*, PDP '08, pages 301–310, Washington, DC, USA, 2008. IEEE Computer Society.

[58] Gregorio Quintana-Ortí, Francisco D. Igual, Enrique S. Quintana-Ortí, and Robert A. van de Geijn. Solving dense linear systems on platforms with multiple hardware accelerators. *SIGPLAN Not.*, 44(4):121–130, February 2009.

[59] E. Agullo, C. Augonnet, J. Dongarra, M. Faverge, J. Langou, H. Ltaief, and S. Tomov. Lu factorization for accelerator-based systems. In *Computer Systems and Applications (AICCSA), 2011 9th IEEE/ACS International Conference on*, pages 217–224, Dec 2011.

[60] E. Agullo, C. Augonnet, J. Dongarra, M. Faverge, H. Ltaief, S. Thibault, and S. Tomov. Qr factorization on a multicore node enhanced with multiple gpu accelerators. In *Parallel Distributed Processing Symposium (IPDPS), 2011 IEEE International*, pages 932–943, May 2011.

[61] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, A. Haidar, T. Herault, J. Kurzak, J. Langou, P. Lemarinier, H. Ltaief, P. Luszczek, A. YarKhan, and J. Dongarra. Flexible development of dense linear algebra algorithms on massively parallel architectures with dplasma. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*, pages 1432–1441, May 2011.

[62] Emmanuel Agullo, Jim Demmel, Jack Dongarra, Bilel Hadri, Jakub Kurzak, Julien Langou, Hatem Ltaief, Piotr Luszczek, and Stanimire Tomov. Numerical linear algebra on emerging architectures: The plasma and magma projects. *Journal of Physics: Conference Series*, 2009.

[63] Field G.Van Zee, Ernie Chan, Robert A.van de Geijn, Enrique S. Quintana-Ortí, and Gregorio Quintana-Ortí. The libflame library for dense matrix computations. *Computing in Science Engineering*, 11(6):56–63, Nov 2009.

[64] X. Lacoste, M. Faverge, P. Ramet, S. Thibault, and G. Bosilca. Taking advantage of hybrid systems for sparse direct solvers via task-based runtimes. In *Parallel Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, pages 29–38, May 2014.

[65] Emmanuel Agullo, Alfredo Buttari, Abdou Guermouche, and Florent Lopez. Multifrontal qr factorization for multicore architectures over runtime systems. In Felix Wolf, Bernd Mohr, and Dieter an Mey, editors, *Euro-Par 2013 Parallel Processing*, volume 8097 of *Lecture Notes in Computer Science*, pages 521–532. Springer Berlin Heidelberg, 2013.

[66] Atsushi Kawai, Toshiyuki Fukushige, Junichiro Makino, and Makoto Taiji. Grape-5: A special-purpose computer for n-body simulations. *Publications of the Astronomical Society of Japan*, 52(4):659–676, 2000.

[67] Andrew W Appel. An efficient program for many-body simulation. *SIAM Journal on Scientific and Statistical Computing*, 6(1):85–103, 1985.

[68] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

[69] Klaas Esselink. A comparison of algorithms for long-range interactions. *Computer Physics Communications*, 87(3):375–395, 1995.

[70] Josh Barnes and Piet Hut. A hierarchical o (n log n) force-calculation algorithm. 1986.

[71] Leslie Greengard and Vladimir Rokhlin. A fast algorithm for particle simulations. *Journal of computational physics*, 73(2):325–348, 1987.

[72] Barry A Cipra. The best of the 20th century: editors name top 10 algorithms. *SIAM news*, 33(4):1–2, 2000.

[73] Michael S Warren and John K Salmon. A parallel hashed oct-tree n-body algorithm. In *Proceedings of the 1993 ACM/IEEE conference on Supercomputing*, pages 12–21. ACM, 1993.

[74] Guy M Morton. *A computer oriented geodetic data base and a new technique in file sequencing*. International Business Machines Company, 1966.

[75] D. Hilbert. Uber die stetige abbildung einer linie auf ein flächenstuck. *Mathematische Annalen 38*, pages 459–460, 1891.

[76] Nico Reissman, Jan Christian Meyer, and Magnus Jahre. A study of energy and locality effects using space-filling curves. In *Parallel & Distributed Processing Symposium Workshops (IPDPSW), 2014 IEEE International*, pages 815–822. IEEE, 2014.

[77] Bongki Moon, Hosagrahar V Jagadish, Christos Faloutsos, and Joel H. Saltz. Analysis of the clustering properties of the hilbert space-filling curve. *Knowledge and Data Engineering, IEEE Transactions on*, 13(1):124–141, 2001.

[78] Paul M Campbell, Karen D Devine, Joseph E Flaherty, Luis G Gervasio, and James D Teresco. Dynamic octree load balancing using space-filling curves. *Williams College Department of Computer Science, Technical Report CS-03*, 1:68, 2003.

[79] Herman Haverkort. An inventory of three-dimensional hilbert space-filling curves. *arXiv preprint arXiv:1109.2323*, 2011.

[80] Pierre Fortin. *Algorithmique hiérarchique parallèle haute performance pour les problèmes à N-corps*. PhD thesis, Université Sciences et Technologies-Bordeaux I, 2006.

[81] S. Jamet, N. Rougemaille, J.-C. Toussaint, and O. Fruchart. Head-to-head domain walls in one-dimensional nanostructures: an extended phase diagram ranging from strips to cylindrical wires. *ArXiv e-prints*, December 2014.

[82] Etcheverry Arnaud, Blanchard Pierre, Dupuy Laurent, and Olivier Coulaud. OptiDis: a MPI/OpenMP Dislocation Dynamics Code for Large Scale Simulations. In *The 7th MMM International Conference on Multiscale Materials Modeling*, Berkeley, United States, October 2014.

[83] Pierre BLANCHARD, Arnaud Etcheverry, Olivier Coulaud, Laurent Dupuy, and Marc Blétry. OptiDis: Toward fast anisotropic DD based on Stroh formalism. International Workshop on DD simulations, December 2014. Poster.

[84] M. Messner, B. Bramas, O. Coulaud, and E. Darve. Optimized M2L Kernels for the Chebyshev Interpolation based Fast Multipole Method. *ArXiv e-prints*, October 2012.

[85] Pierre Blanchard, Olivier Coulaud, and Eric Darve. Fast hierarchical algorithms for generating Gaussian random fields. Research report, Inria Bordeaux Sud-Ouest, November 2015.

[86] Lexing Ying, George Biros, Denis Zorin, and Harper Langston. A new parallel kernel-independent fast multipole method. In *Supercomputing, 2003 ACM/IEEE Conference*, pages 14–14. IEEE, 2003.

[87] Tsuyoshi Hamada, Tetsu Narumi, Rio Yokota, Kenji Yasuoka, Keigo Nitadori, and Makoto Taiji. 42 tflops hierarchical n-body simulations on gpus with applications in both astrophysics and turbulence. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC '09, pages 62:1–62:12, New York, NY, USA, 2009. ACM.

[88] A. Chandramowlishwaran, S. Williams, L. Oliker, I. Lashuk, G. Biros, and R. Vuduc. Optimizing and tuning the fast multipole method for state-of-the-art multicore architectures. In *Parallel Distributed Processing (IPDPS), 2010 IEEE International Symposium on*, pages 1–12, April 2010.

[89] Qi Hu, Nail A. Gumerov, and Ramani Duraiswami. Scalable fast multipole methods on distributed heterogeneous architectures. In *Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '11, pages 36:1–36:12, New York, NY, USA, 2011. ACM.

[90] Rio Yokota and Lorena A Barba. Treecode and fast multipole method for n-body simulation with cuda. *GPU Computing Gems Emerald Edition*, page 113, 2011.

[91] Ilya Lashuk, Aparna Chandramowlishwaran, Harper Langston, Tuan-Anh Nguyen, Rahul Sampath, Aashay Shringarpure, Richard Vuduc, Lexing Ying, Denis Zorin, and George Biros. A massively parallel adaptive fast multipole method on heterogeneous architectures. *Commun. ACM*, 55(5):101–109, May 2012.

[92] Dhairya Malhotra and George Biros. Pvfmm: A parallel kernel independent fmm for particle and volume potentials. *Communications in Computational Physics*, 18(03):808–830, 2015.

[93] Hatem Ltaief and Rio Yokota. Data-driven execution of fast multipole methods. *Concurrency and Computation: Practice and Experience*, 26(11):1935–1946, 2014.

[94] T Abboud, M Pallud, and C Teissedre. Sonate: a parallel code for acoustics nonlinear oscillations and boundary-value problems for hamiltonian systems. Technical report, Technical report, http://imacs. xtec. polytechnique. fr/Reports/sonate-parallel. pdf, 1982.

[95] Fang Q Hu. An efficient solution of time domain boundary integral equations for acoustic scattering and its acceleration by graphics processing units, 19th aiaa. In *CEAS AEROACOUSTICS CONFERENCE, Chapter DOI*, volume 10, pages 6–2013, 2013.

[96] Toru Takahashi. An interpolation-based fast-multipole accelerated boundary integral equation method for the three-dimensional wave equation. *Journal of Computational Physics*, 258:809 – 832, 2014.

[97] Johannes Tausch. A fast method for solving the heat equation by layer potentials. *Journal of Computational Physics*, 224(2):956 – 969, 2007.

[98] Patrick R Amestoy, Iain S Duff, Jacko Koster, and Jean-Yves L'Excellent. Mumps: A multifrontal massively parallel solver. *ERCIM News*, 50:14–15, 2002.

[99] Arnaud Etcheverry. *Vers la simulation en dynamique des dislocations à grande échelle*. PhD thesis, Université Sciences et Technologies-Bordeaux I, 2015.

[100] Christopher A White and Martin Head-Gordon. Rotating around the quartic angular momentum barrier in fast multipole method calculations. *The Journal of Chemical Physics*, 105(12):5061–5067, 1996.

[101] Holger Dachsel. Fast and accurate determination of the wigner rotation matrices in the fast multipole method. *The Journal of chemical physics*, 124(14):144115, 2006.

[102] A. A. Ergin, B. Shanker, and E. Michielssen. Fast analysis of transient acoustic wave scattering from rigid bodies using the multilevel plane wave time domain algorithm. *The Journal of the Acoustical Society of America*, 107(3), 2000.

[103] Guillaume Sylvand. *La méthode multipôle rapide en électromagnétisme. Performances, parallélisation, applications*. PhD thesis, Ecole des Ponts ParisTech, 2002.

[104] Eric Darve. *Méthodes multipôles rapides: Résolution des équations de Maxwell par formulations intégrales*. PhD thesis.

[105] Pascal Havé Eric Darve. A fast multipole method for maxwell equations stable at all frequencies. *Philosophical Transactions: Mathematical, Physical and Engineering Sciences*, 362(1816):603–628, 2004.

[106] Eric Darve. The fast multipole method: numerical implementation. *J. Comput. Phys*, 2000.

[107] A.Arif Ergin, Balasubramaniam Shanker, and Eric Michielssen. Fast evaluation of three-dimensional transient wave fields using diagonal translation operators. *Journal of Computational Physics*, 146(1):157 – 180, 1998.

[108] Matteo Frigo and Steven G Johnson. Fftw: An adaptive software architecture for the fft. In *Acoustics, Speech and Signal Processing, 1998. Proceedings of the 1998 IEEE International Conference on*, volume 3, pages 1381–1384. IEEE, 1998.

[109] Daniele Funaro. *Polynomial approximation of differential equations*, volume 8. Springer Science & Business Media, 1992.

[110] Lloyd N Trefethen and David Bau III. *Numerical linear algebra*, volume 50. Siam, 1997.

[111] Steven C Chapra and Raymond P Canale. *Numerical methods for engineers*, volume 2. McGraw-Hill, 2012.

[112] William H Press. *Numerical recipes 3rd edition: The art of scientific computing*. Cambridge university press, 2007.

[113] Alexander D Poularikas. *Handbook of formulas and tables for signal processing*, volume 13. CRC Press, 1998.

[114] Intel. *Intel Intrinsics Guide*. https://software.intel.com/sites/landingpage/IntrinsicsGuide/, october 2015 edition.

[115] Intel. *Intel® 64 and IA-32 Architectures Software Developer Manuals*. http://www.intel.com/content/www/us/en/processors/architectures-software-developer-manuals.html, 055 edition.

[116] AMD. *AMD® Technical Documents and Architecture Programmer's Manual*. http://support.amd.com/en-us/search/tech-docs.

[117] Oleg Mishchenko. Optimizing cache behavior of ray-driven volume rendering using space-filling curves. Master's thesis, Stony Brook University, 2006.

[118] Ananth Grama. *Introduction to parallel computing*. Pearson Education, 2003.